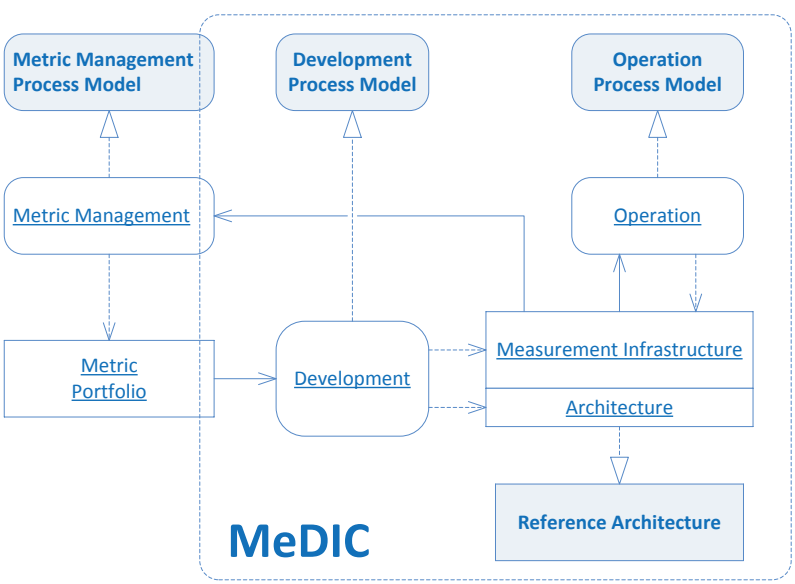
Matthias Vianden

Systematic Metric Systems Engineering

Reference Architecture and Process Model



Aachener Informatik-Berichte,
Software Engineering

Band 26

Hrsg: Prof. Dr. rer. nat. Bernhard Rumpe
Prof. Dr. rer. nat. Horst Lichter

Systematic Metric Systems Engineering: Reference Architecture and Process Model

Der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University vorgelegte Dissertation zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften

vorgelegt von

Diplom-Informatiker
Matthias Vianden
aus
Aachen

Abstract

In the recent past, the research community contributed a considerable amount of work to extend the understanding of the theoretical foundations of metric systems. However, a dedicated approach for engineering of metric systems is still missing. As a result, they are often developed chaotically. This thesis introduces MeDIC – a dedicated metric systems engineering approach, which fills this gap. MeDIC supports flexible conception, design, construction, and operation of metric systems. The approach is based on two pillars: the *MeDIC process model* and the *MeDIC reference architecture*. They integrate software engineering best practices, emerging concepts, and well-established metric-related standards and techniques. The MeDIC reference architecture provides technical guides with a layered architecture blue-print of loosely interconnected micro-services. The MeDIC process model provides ready-to-use process elements and artifacts (fragments), which drastically ease the setup of a specific engineering process. The reference architecture and process model are based on formal foundations, which provide additional benefits for conceptual analysis of metrics systems. Various field studies, in cooperation with multiple industry partners, were used to evaluate the approach. This thesis provides insight into three selected field studies, which utilize various aspects of MeDIC in industrial environments. The evaluation shows the practical application, usefulness, and efficiency of MeDIC. Challenges associated with the development and operation of metric systems in industrial environments can thus be overcome by MeDIC. As a result, the engineering of better, more reliable, and sustainable metric systems is possible.

Kurzfassung

In der Vergangenheit wurde vermehrt an den theoretischen Grundlagen der (Software-) Metriken gearbeitet. Das allgemeine Verständnis hat sich seitdem stark weiterentwickelt und neben den theoretischen Grundlagen hat sich auch das Verständnis des Metrikmanagement weiterentwickelt. Hieraus entwickelte sich ein etablierter Stand der Technik, welcher sich in diversen Standards widerspiegelt (ISO 15939, CMMI MA). Es fehlt allerdings immer noch ein spezieller Engineering-Ansatz für Metriksysteme und die damit verbundene Messinfrastruktur, was dazu führt, dass diese oft chaotisch entwickelt werden. Der in dieser Arbeit vorgestellte flexible Engineering-Ansatz MeDIC schließt diese Lücke und adressiert sowohl die klassischen Phasen des Software-Engineerings: Konzeption, Entwurf und Konstruktion, als auch Betrieb von Metriksystemen. MeDIC besteht aus zwei fundamentalen Teilen: dem MeDIC-Prozessmodell und der MeDIC-Referenzarchitektur. Diese integrieren best-practices der Software-Entwicklung, moderne Konzepte und etablierte Techniken und Standards im Bereich der Metriken miteinander. Die MeDIC-Referenzarchitektur stellt eine Blaupause einer geschichteten Architektur von lose gekoppelten Micro-Services zur Verfügung und erhöht damit das Verständnis der technischen Konzepte. Das MeDIC-Prozessmodell enthält fertig verwendbare Prozessbausteine und Artefakte (-Fragmente), welche den Aufbau eines dedizierten Engineering-Prozesses drastisch vereinfachen. Die Referenzarchitektur und das Prozessmodell sind mit einer formalen Basis untermauert, welche zusätzlich die Analyse von Metriksystemen auf einem theoretischen und konzeptuellen Niveau ermöglicht. MeDIC wurde in zahlreichen industriellen Feldstudien evaluiert. Diese Arbeit stellt drei ausgewählte Feldstudien vor, welche unterschiedliche Aspekte von MeDIC im industriellen Umfeld verwenden. Hierdurch wird die praktische Anwendbarkeit, Nützlichkeit und Effektivität des Ansatzes gezeigt. MeDIC hilft, viele der praktischen Probleme beim Entwickeln und Betreiben von Metriksystemen im industriellen Umfeld zu überwinden. In der Zukunft können diese Systeme mit Hilfe von MeDIC demnach besser, zuverlässiger und nachhaltiger entwickelt werden.

Contents

List of Figures

Part I.

Introduction and Foundations

1 Introduction

Gathering process and product data are central necessities for a mature software development organization. The gathering and measurement of these data is defined by metrics. Consequently, the IEEE software glossary defines metrics as "a quantitative measure of the degree to which a system, component, or process possesses a given attribute" [Ele90]. In this thesis we follow the operation focused definition for metric from the ISO 15939 as the specification of a measurement (the measurement approach and process) [ISO07].

Process improvement models such as CMMI or SPICE encourage software development organizations to build up abilities to systematically utilize metrics and measure the quality of the development processes and software systems as one of the basic steps to higher maturity [Tea10]. By analyzing metrics, process managers are able to identify processes that contribute to project success or failure [HL11, Dyb05, Kir01]. Metrics provide the harness for organization strategies in the sense of performance measurement systems [Mel04]. Furthermore, metrics provide the basis for solid project management [WGT07, Mul13].

Even though metrics are important, the research community and industry agrees that it is often difficult to find the *right* metrics and provide *good* measurements [Fen94, Bas92]. Quality standards like the ISO 25000 series try to overcome this problem by providing a broad number of metrics embedded in a rich quality model to guide metric selection [ISO03, ISO14]. Kilidar et al., however, question the usefulness of the standard. They conclude that "the standard was ambiguous in meaning, incomplete with respect to quality characteristics and overlapping with respect to measured properties" [AKCK05]. Another problem of quality models is that they are rarely practically applicable because a lot of the metrics are not calculable. Wagner et al. bridge this problem with their QUAMOCO approach which uses measurements from a broad variety of measurement instruments to feed the calculation inside their quality model [WGH+15, WLH+12, WBD+10]. Thus, enabling practical usage of the quality model. However, most stakeholders are interested in very specific aspects of the software development process or product. Hence, it is often necessary to define suitable metrics specific to the questions at hand.

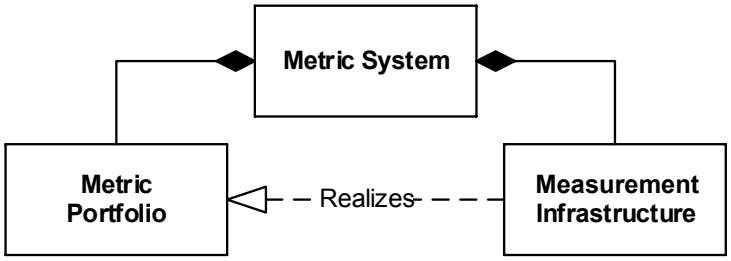


Figure 1.1.: Metric system decomposition

Figure 1.1 shows our differentiation of metric systems into two parts: the metric portfolio which contains the definitions for the metrics and indicators[1] and the measurement infrastructure which contains the actual software systems that calculate the metrics and provide the visualizations specified by the indicators. In this thesis, in accordance with international standards and related literature, we therefore define:

Metric portfolio as a set of definitions for metrics and indicators (see section 2.1 for further details).

Measurement infrastructure as all the technical components that realize and are required for the realize of metrics and indicators.

Metric system as a metric portfolio and a measurement infrastructure which realizes the metric portfolio.

Most existing approaches focus on the management and changes to the metric portfolio [ISO07, Tea10, USE, FP98, CK94]. Some approaches focus on the measurement infrastructure [HM04, Hei08, YDSN10]. Very few approaches combine the two sides and from our knowledge no approach exists that explicitly addresses the realization from metrics in the metric portfolio in parts of the measurement infrastructure (the development). From our experience most of the measurement infrastructures that implement specific metric systems are poorly build [VLJ13, VLS13]. They are often implemented using spreadsheet applications plus additional scripts to implement the measurement of the base data from different sources [SMN09, VLJ13, Pia07]. This often results in poorly maintainable solutions, which are not able to evolve with constant changes of the environment [VLJ13].

This thesis presents MeDIC[2] a dedicated metric systems engineering approach to overcome these problems. MeDIC consists of two main parts: a reference architecture for measurement infrastructures founded on the ideas of enterprise application integration, microservices, and separation of concerns as well as a metric systems engineering process model that integrates metric specific tools like GQM for requirements elicitation and software engineering best practices like prototyping, incremental and iterative development, and defined staging and release of the software tools.

[1] Again, we use the ISO 15939[ISO07] definition for “metric” and “indicator”.

[2] MeDIC is an acronym for Metric systems Development, Infrastructure, and Concepts. These are the key parts of the engineering approach.

A thesis should not repeat over and over what others wrote many times before, what is well known, and what is available through various literature. It should be build on these foundations. Therefore, we do not provide a separate introduction into metric and measurement theory and processes. This includes special metrics and measurement aspects like a description of the measurement process from the ISO 15939 and CMMI Measurement and Analysis [ISO07, Tea10] as well as general aspects such as working with and defining metrics, which is included in most popular SE text books [Som11, LL13], often taught in universities, and included in various theses[3]. This thesis focus on the challenges when building measurement infrastructures and defining metrics in different environments and tackles these with a dedicated engineering approach for metric systems.

Thesis Structure

This thesis contains three central parts covering the main aspects of our engineering approach. The details of our reference architecture for measurement infrastructures are presented in part II. Part III then contains the process model for metric systems development and operation. The evaluation of these two aspects based on our tooling and selected field studies is presented in part IV. This core of the thesis is surrounded by the conclusion and outlook part V and the introduction and foundations in part I. This first part contains the central metric systems engineering challenges, our research contribution, and a quick introduction to the research field as well as the conceptual foundations including the formal foundations for MeDIC.

[3] Very good introductions to these topics are provided in the PhD thesis of Jens Heidrich [Hei08] as well as the PhD thesis of Martin Kunz [Kun09].

1.1. Metric Systems Engineering Challenges

The analysis of metric systems engineering in practice presented in this section is based on our experiences from building metric systems with different cooperation partners over the past 8 years [Via08, Hut13, Cho11, Han12, VLJ13, Eve10, VBR09]. We like to differentiate our experience into large software development companies and small/medium sized software development companies because the environment for projects in these categories drastically differs from each other. The definitions for software company sizes differs a lot over various studies. For this thesis we define small software companies as companies with less than 40 (full time equivalent) employees, medium sized companies range from 41 until 250 employees, and large companies employ more than 250 people. We also did a lot of work focusing on research projects. However, from our experience research projects roughly behave like small/medium sized companies. Hence, we do not provide an additional section for our experience in research projects.

The focus of this section is to analyze the status-quo of the engineering approaches for metric systems at our cooperation partners before we started working with them as well as best practices from our experiences. From this we identify a set of common challenges that need to be tackled when engineering metric systems.

1.1.1. Large Software Development Companies

Our experience witch large software development companies is based on our cooperation with two large IT service providers for insurance companies located in Germany. One of these companies is CMMI level 3 certified. During the last 30 years both companies produced a broad legacy of systems which support all kinds of processes in the insurance domain. This leads to a lot of challenges maintaining and enhancing the existing systems. In both companies this is supported by company wide processes and tools. Also they based their processes on well established process models like the rational unified process (RUP). Even though they currently investigate the use of agile software development ideas most of the existing projects are based on traditional (PM BoK) project management. This section only provides a summary of the problems and challenges that we faced while working with our cooperation partners [4]. The following list states the three main challenges that we found with the existing metric system and dashboards in large software development companies.

Purposefulness The metrics need to serve a purpose i.e. they need to answer questions (information needs) from metric customers. Furthermore, all roles[5] involved with developing and operating the metric system need to be addressed as well.

Unfortunately, our experience showed that the metric systems often provide unnecessary information and do not answer crucial questions of the metric customers. Therefore, an engineering approach for metric systems needs to include a thorough

[4] A more detailed analysis of our experience is available in our paper for the first QuASoQ workshop 2013 in Bangkok [VLHH13] as well as the diploma thesis from Andrea Hutter [Hut13].

[5] See section 9.3 for more information on the different stakeholders and roles.

requirements phase to identify the information needs of the metric customers. Furthermore, it needs to include constant validation of the proposed metrics with the metric customers.

Reuse Both large cooperation partners run approximately 90 projects per year. Each of the 90 project managers have similar (yet slightly different) information needs focusing on the status of the project. This leads to the development of templates for the project management dashboards which are then used by the project managers to create the project dashboard. Unfortunately, the template were arbitrarily instantiated and engineered because the companies do not define systematic metric reuse concepts. Problems emerged for example with the metric definitions and metric documentations when the template was instantiated and arbitrarily altered to fit other needs. These problems then escalated because the CMMI level 3 requirements for tailoring documentation and definition were not met. Hence, an engineering approach for metric systems needs to include a systematic approach for metric reuse and reuse in measurement infrastructure components like dashboards.

Usability We noticed that the interpretation of the metrics is often hard for people that only temporarily work with them. Furthermore, the configuration of the metrics and selection of the metrics was far from intuitive. These two aspects lead to misinterpretations either because the metric was not understood, was wrongly configured, or wrongly selected. All of this renders the metric system effectively useless. Therefore, an engineering approach for metric systems needs to enforce usability of the metric tools and needs to include metric documentation.

Environmental changes Environmental changes include process and organizational changes, tool changes, and technological changes. All these changes influence the metric system. Most of them change the information needs of the metric customers and some of them influence the data collection mechanism in the measurement infrastructure. Hence, a flexible engineering approach for metric systems needs to address these changes pro actively.

From our experience, providing flexibility and enabling metric evolution is the most crucial aspect that we identified from our work with large cooperation partners.

1.1.2. Small and Medium Software Development Companies

During the period between 2007 and 2014 we were able to build several metric systems with multiple small and medium sized software development companies [Via08, VBR09, Han12, Hut13]. They produce specific software products supporting smaller markets (energy contract management, license management, leasing calculation, and price information systems). The company size ranges from 10 to 250 employees. Unfortunately, we did not have the chance to work with an IT consulting company or IT infrastructure provider.

Contrasting the large companies from above these companies did not have a company wide metric system or metric program. However, most of them did monitor the working hours of their employees. This information was then used to keep track of the effort for specific working packages or projects. Sometimes the project managers did use specific reports from their requirements management systems (mostly ticketing systems) to get an overview over the current work status. However, non of the companies that we worked with utilized software metrics. We believe the reasons for this lack of metrics at small and medium sized companies are:

Special Development Environment and Tools Most of the companies used very specific environments to build their software. One of the companies for example uses Delphi to build the clients for their products. These clients connect to oracle databases. The business logic of the applications, however, is spread between these two environments. Some of the business methods are located in the Delphi clients; some are located in PL/SQL stored procedures in the database. Therefore, we needed to develop a specific measurement tool that was able to provide statical analyzes for code in both languages. It also needed to analyze the links between the two systems [Via08, VBR09]. However, building and maintaining such a specific metric system and measurement infrastructure is costly and not necessarily something the management of small and medium sized companies want to invest in.

Special Information Needs require special Metrics Each of the different stakeholders from each company had very specific information needs that needed to be answered. Unfortunately, the answers to these specific questions are very rarely provided by standard measurement tools. For example, the company from the example above was confronted with multiple questions related to the complexity of their software. More specifically, questions related to source code complexity based on a prediction of the test effort. Even though, some scientific work on this topic exists, non fitted the need of the company. Hence, new metrics, which predict the test effort from static source code analysis needed to be specified and implemented [VLR09].

Another company uses PHP to build a large system over the last 10 years. The GQM workshops showed that classic source code metrics like LOC or CC do not provide answers for the questions that the managers and quality assurance staff had. For example, they were facing the challenge to enforce object orientation over imperative procedures located near the corresponding HTML code. Therefore, again we needed to implement specific metrics to answer this question [Han12].

Technology and tools change frequently Again contrasting the large companies one of the characteristics of small companies is that they are able to quickly change tools and technology if they feel they need to. This not only includes standard tools like issue tracker / ticketing systems but also specific tools like time sheet software. Also, development methodologies constantly change. We observed multiple methodology shifts from procedural imperative development towards object

oriented framework-based development. Such changes always imply changes in the metric system.

No company wide standard Some of the companies did have multiple branches that develop different software for similar or sometimes different markets. The branches, however, often did not share similar technologies or development metaphors. This, again, induced a broad variety amongst the information needs of even similar stakeholders in multiple branches. Consequently, this leads to a broad variety in the metric system as well.

Summarizing the brief description of our experiences from above a metric systems engineering approach for small and medium sized companies needs to be very flexible to support the large variety amongst the different information needs and needs to support constant change (evolution) of the metric system. The measurement infrastructure of the metric system needs to support this by providing clear separation between the different central tasks: measurement, calculation, and visualization. Again, this section only summarized our findings[6].

We also analyzed the use and usefulness for metric systems in a broad variety of research projects (mostly software engineering research) [Hut13] and build dedicated measurement infrastructures for software engineering research projects [Ott13]. In essence our experience with engineering metric systems for (software) research projects mirrors our findings for small an medium sized companies. Hence, we do not address the challenges and needs for research projects explicitly in this thesis.

1.1.3. Main Challenges

Our field studies generally show a lack of solid engineering when developing metric systems. Thus, the overall goal of a systematic metric systems engineering approach should be to support the development, and operation of metric systems which are successful (e.g. cost efficient) in the long run. Summarizing our experience from above and adding to the list in our existing publications (see [VLJ13]) we extracted the following three main challenges:

C1 - Flexibility: All of our field studies registered environmental changes to the metric system. Change sources were heterogeneously distributed across different aspects. They included the utilized tools and processes of the company for example. Thus, these changes need to be reflected in the metric system by the means of changed metrics and according changes in the measurement infrastructure. These maintenance related tasks make up a majority of the overall costs and effort of a

[6] A more detailed description of the challenges with building metric systems at small companies as well as a detailed description about the engineering of metric systems using our engineering approach can be found at the bachelors thesis from Christian Hans [Han12]. Also, again Andrea Hutter provides an analysis of the metric needs of small companies in her diploma thesis [Hut13]. Finally, the diploma thesis of Matthias Vianden also contains important information on the challenges and engineering of metric systems at medium sized companies [Via08].

software system throughout its lifetime [Som11, BR00, BDKZ93]. Hence, aiming for flexible metric systems must be one of the mayor goals of the engineering approach.

C2 - Solid and Specific (Software) Engineering: Similar to changes, all of our field studies indicated a lack of adhering to solid software engineering practices when developing metric systems and measurement infrastructures in the field. This lack can also be observed with typical prototypes developed in research environments and tools out in the field. Other fields like development of web applications / web pages, information systems, or databases reflected equivalent lack with specific software engineering approaches [SGN08, CCP07, Bah09, ST]. We believe that the development of metric systems similarly requires a dedicated metric systems engineering approach. This approach then needs to reflect current software engineering best practices.

C3 - Usefulness and Usability: Like many others [Hei08, Kun09, DLGP10, NV01, HMO08, CSS09] we noticed that most of the metric systems did not answer the specific information needs of the metric customers. Which renders most of them useless and degenerate them to a pure reporting tool. Some problems that we observed were the lack of dedicated (metric) requirements analysis techniques like GQM, the lack of quality assurance techniques like prototyping for the requirements, missing or unusable documentation for the metrics, and missing integration between metric specification and measurement infrastructure and dashboards. Additionally, we often observed a lack in usability of the metric specification and dashboard tools. They were often very much driven by technology like data warehouse front ends to specify ETLs or spreadsheet applications. Furthermore, the specification and documentation of the metrics (if present) often lacks the link to the information needs of the customers that are answered by the specific metric.

1.1.4. Summary

This section presented an overview of metric systems engineering challenges that we faced when analyzing metric systems engineering in the field. Most importantly we noticed a lack of solid software engineering practices when developing metric systems. Figure 1.2 depicts the situation like we faced it in the industry. Metric management processes, based on metric management process models like the ISO 15939, change the items in the metric portfolio. New metrics are added and existing metrics evaluated by usage feedback from the measurement infrastructure. The development, maintenance, and operation of the measurement infrastructure from the metric portfolio, however, is performed chaotically, unplanned, and far from solid software engineering. Furthermore, we noticed that the architecture of the measurement infrastructure is not explicitly engineered as well.

The success of measurement best practices, like GQM for metric requirements engineering, and success of specific engineering approaches, for example for quality aware web engineering [CCP07], indicates the benefits for a dedicated metric systems engineering approach. Still, such an engineering approach needs solid foundations. As

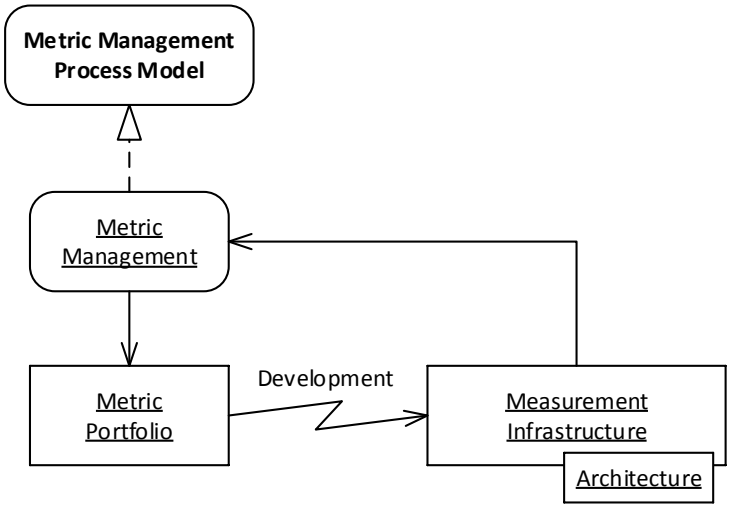


Figure 1.2.: Development gap in metric systems engineering.

such, the condensed list of top level challenges for the engineering approach from our field study experiences presented above are used to formulate a short list of top level requirements in the next section.

1.2. Top-Level Requirements

The challenges presented in the last section provide an incentive for a dedicated metric systems engineering approach. However, they are still quite vague and unspecific. Thus, this section derives a list of top level requirements for solid metric systems engineering from the challenges and our field study experiences. We address the metric systems development process and measurement infrastructure separately because they aim on different part of an engineering approach (method versus tool). With each requirement we list the index of the challenge from above that is addressed with the requirement.

Metric System Development Process Requirements

Our hands-on experience shows that metric system development is often chaotic and common software engineering best practices are ignored. Even metric initiatives launched in companies that apply good development processes and best practices for the development of high quality and successful software seem to ignore all of this when it comes to the development of metric systems. Hence, the following list provides a condensed set of requirements for the development of metric systems from a process perspective:

ReT-P1 The process model needs to clearly define the roles and responsibilities for the different stakeholders associated with the development and operation of a metric system. (C2, C3)

ReT-P2 The process model needs to include developers and operators. (C2)

ReT-P3 The process model needs to include activities to continuously evaluate the information needs underlying each metric. (C2, C3)

ReT-P4 The process model needs to encourage the development of diverse metric systems that include specific dashboards and analysis tools which address the different roles and their specific information needs differently. (C1)

ReT-P5 The process model needs to be founded on software engineering best practices. Most importantly, it should include prototyping activities to validate the requirements for the metric system before investing time and money into actually building a measurement infrastructure. (C2)

ReT-P6 The process model needs to embrace flexibility by including iterative and incremental development. (C1)

Measurement Infrastructure Requirements

From our brief description above an obvious requirement for the measurement infrastructure is its flexibility. The following list of requirements further specifies flexibility of the measurement infrastructure by addressing specific aspects of the infrastructure:

ReT-I1 The measurement infrastructure needs to support easy integration of heterogeneous data sources as the basis for different metrics and visualizations. (C1)

ReT-I2 The measurement infrastructure needs to enable fast and up-to-date recognition and update of the metrics on a change in a data source. (C1, C3)

ReT-I3 The measurement infrastructure needs to clearly separate system integration, metric calculation and visualization. (C1, C2, C3)

ReT-I4 The measurement infrastructure needs to be robust to avoid a complete system failure if a small part of the system fails. (C2, C3)

ReT-I5 The measurement infrastructure needs to be isolated in the sense that a failure of the infrastructure does not result in a failure in the data source. (C2, C3)

ReT-I6 The measurement infrastructure needs to not force one central data schema or one database technology. This avoids schema-mapping problems and support flexible isolated evolution of different data schemata. It also enables to select the database (technology) that best fits the specific need. (C1)

ReT-I7 The measurement infrastructure needs to support the evolution of metrics, measurements, and visualizations. (C1, C2)

ReT-I8 The measurement infrastructure needs to offer support for dedicated operation tools. (C2)

1.2.1. Literature Analysis

The top level requirements from above are based on the challenges that we experienced when analyzing and developing metric systems in our industrial and research projects. This section provides an analysis of various best practices, guidelines, and critical success factors from the literature based on our requirements. The goal is to find additional support for our requirements in the literature. We also discuss missing support or different opinions on certain parts.

Ciolkowski et al. presented a comprehensive list of guidelines when building measurement systems in their 2008 article "Practical Guidelines for Introducing Software Cockpits in Industry" [CHM08]. Another very rich and valuable resource is the measurement maturity model by Diaz-Ley et al. [DLGP08b, DLGP10] defined between 2008 and 2010. Looking back even further Niessink et al. investigated key success factors for measurement programs in 2001 [NV01]. Also in 2008 Harjumaa, Markkula, and Oivo published a meta analysis on measurement program success factors that also includes the list by Niessink et al. [HMO08]. Their success factors, however, are not indexed. Hence, we reference the original lists and mention the meta analysis when needed. In 2009 Coman, Sillitti, and Succi published a case study on using an automated in-process software engineering measurement and analysis system in an industrial environment at

the ICSE [CSS09]. The paper starts with another meta analysis on critical success factors for establishing software measurement programs. Again, the list is not indexed and we refer to the items when needed.

From our experience, most measurement success factors and guidelines are easily found or known. Hence, organizations typically follow the guidelines and success factors when starting a metric initiative. For example: they almost always get management support. Unfortunately, from our experience the software development part is often ignored and tools are implemented in an ad-hoc manner. This lack of solid software engineering for the tools and integration between the metrics, measurement infrastructure, and tools is also reflected in the standards. The ISO 15939 measurement process for example does only reflect the pure measurement related phases: "Establish and sustain measurement commitment", "Plan the measurement process", "Perform the measurement process", and "Evaluate measurement". Hence, our requirements focus more on the neglected development part of the metric process. In the following paragraphs the key success factors for measurement programs from Niessink et al. are prefixed by an "I" and an "E", reflecting the original index, whereas the guidelines from Ciolkowski et al. are prefixed by a "G", again similar to the original work. We refer to the two references simply as *the lists* because success factors and guidelines are on different maturity levels.

Process Requirements

One success factor that always shows up is the need for a well planed process and good project management when initiating a metric initiative (I2 and G10-G14). This is mostly reflected in a very thoroughly planed measurement process. Sadly, as stated before we did not find any publication that also included the software development part of the measurement initiative. However, this process and project management emphasize provides a valid support for our process requirements ReT-P1 to ReT-P6.

Investigating these factors in more detail, the requirement ReT-P1 for clearly defined development roles is supported by I13 (metric expert at company) and G2 (address all stakeholders). The work by Coman et al. also lists a "Dedicated team for measurement" as a crucial success factor. The requirement ReT-P2 to include developers and operators in the process is obvious from common SE knowledge but not included in any measurement related guideline or best practice. The continuous evaluation of information needs listed as requirement Ref-P3 is supported by I12 (constantly improve the measurement program), E4 (monitor the implemented changes), and G23 (conduct post-mortem analysis). Requirement ReT-P4 to encourage diversity in the metric system is supported by I6 (usefulness of metric data), G6 (customize solution), and G9 (respect heterogeneity). ReT-P5 to base the development process on common SE best practices again should be obvious. However, we like to list it explicitly because it is not listed in any of the other lists. Strangely, non of the lists explicitly lists prototyping as an important factor because it proved to be very successful and useful in our cases and is listed as a crucial success factor in other areas [BBHM95]. Very short iterations are a suitable substitute for prototyping in agile software development. However, establishing a metric program in a large organization is a lot of effort and short iterations are not possible. We event

found this to be true in smaller organizations. Flexibility as required by ReT-P6 is not included in the lists. We believe the reason for this is that the case studies that support the guidelines and success factors were too short[7] or the companies where too small. However, both of them list incremental development as a crucial success factor: I1 (incremental implementation) and G7 (follow incremental approach).

Infrastructure Requirements

The infrastructure requirements are harder to correlate to the literature because most of the existing work focuses on the measurement process and mostly only touches upon the development and infrastructure aspects. However, Ebert et al. published a book about best practices in software measurement that contains a specific section on measurement infrastructure ([EDBS04], pp. 81-94). They list flexibility of the measurement infrastructure as an important factor: "The system must be able to incorporate new metrics and their interpretations [...] relatively easily". Unfortunately, measurement infrastructures in their work is just a metric database plus additional services. Kunz et al. also investigated measurement infrastructures [Kun09, KSDW06]. The requirements for their measurement infrastructure, however, only contain very abstract goals like "Overcoming of general measurement tool shortcomings" or "Support of corporate measurement programs" as well as non measurement related and from our experience unimportant aspects like "Enable different license models". We discuss their work in greater detail later in this thesis in section 1.4. Additionally to these resources, the need for a specific measurement infrastructure and dedicated tool support is always included in most of the literature. It is also included in the lists as G6 (customize solution) and I11 (use automated data collection tools).

The support for easy integration in requirement ReT-I1 is supported by G8 (integrate into process and tools), G5 (integrate existing data), and I3 (integrate existing metrics material). The requirement ReT-I2 for low latency in the measurement is not explicitly included in the lists. However, it is listed as an important information need for (project) managers in the work from Kunz ("Immediate project review" [Kun09], p. 35). Additionally, the work by Coman et al. lists "Prompt feedback" and "High frequency of data collection" as critical success factors [CSS09]. Low latency increases transparency because changes on the base data is immediately reflected in the measurements. Transparency is listed as an important factor in the lists as G12 (guarantee transparency) and I5 (measurement process transparent [...]). Low latency and specific tool support also increases accessibility as required from G21. A clear separation between the different tasks of a measurement infrastructure as required in ReT-I3 is supported by differentiation between these tasks in the guideline list (G18-G20 and G11). Additionally, these tasks are typically performed by different tools. The requirements ReT-I4 and ReT-I5 for robustness and isolation of the measurement infrastructure are obvious for distributed

[7] If the time frame is too short the chance for the measurement system to reflect changes is very small. However, the organization, development processes, and tools will change if the time frame is long enough. Hence, it is necessary to enforce flexibility of the measurement infrastructure and in the development process.

systems [FT02, Fie00]. Also, these two factors are important quality attributes in common quality models like the ISO 25010:2011 [ISO05]. The requirement ReT-I6 for not relying on a central data schema (and hence not central database) goes against common measurement guidelines. These typically require one central database and hence one central data schema. However, our experience with evolving measurement infrastructures and common database and information system knowledge indicates the benefits of avoiding a central data schema [AS07, BR01, Ber11, Bey07]. Hence, even though the requirement conflicts with typical measurement best practices we believe requirement ReT-I6 is important for long time success of evolving metric systems and corresponding measurement infrastructures. This implies requirement ReT-I7 to support such evolving metric systems. Unfortunately, most of the case studies, field studies, and experience reports in the literature only cover small time frames and small teams[8]. The meta analysis by Harjumaa et al., however, lists the two success factors "Capability to change" and "Constant improvement of the metrics program" as more important than for example "Ensuring integrity of data" [HMO08]. The requirement ReT-I8 for dedicated operation support is very technical and hence not included in the lists from above. However, the need for dedicated operation support, especially for loosely coupled distributed systems, and stronger integration between development and operation is a main claim by DevOPs initiatives [Hüt12].

1.2.2. Summary

This section manifests the challenges from the previews section by forging two specific lists of dedicated top level requirements for a metric systems engineering approach. The requirements address the two crucial parts of a dedicated metric systems engineering approach: the development process and the measurement infrastructure. Even though the requirements and goals are justified by our field study experience we closed the section with a thorough discussion of our requirements based on established literature. The discussion showed the strong support for our requirements from established literature. Also, the discussion again showed the need for a dedicated metric systems engineering approach. Based on this, the following section introduces our research questions and contributions of this thesis based on our experiences, the challenges, and requirements from the previews and this section.

[8] The measurement program in the case study in the 2010 paper from Diaz-Ley et al. was only one year old (start in 2007 and evaluation in 2008) [DLGP10]. The four case studies in the work from Niessink et al. only cover very specific measurement initiatives on a very small time frame as well. They start with a specific question, for example: "Are function points a means to overcome problems with the negotiation of the price of changes with the customer?", which is then tackled with a specific measurement initiative. As soon as the particular question is answered, in the example: "No.", the measurement is over.

Such short measurement initiatives do not need to adhere to environmental changes. Hence, do not need to evolve.

1.3. Research Questions and Contribution

The challenge from our field studies, the top level requirements and several indicators from the literature discussed above imply the need for a dedicated holistic software engineering approach for metric systems. The engineering approach addresses our main research question:

> Q1: How to support flexible, information need driven (goal oriented) engineering (development and operation) of usable metric systems?

An engineering approach according to established software engineering literature needs to include: a method or process component, a tool or technological component, and a notation component. Therefore, we differentiate our main research question into the following three questions:

> Q2,3,4: How to support flexible, information need driven engineering for usable metric systems from the [process, technology, notation] perspective?

Last but not least this thesis addresses the research question (Q5) on how effective and efficient is the engineering approach in practice?

1.3.1. Contribution

The overall contribution of this thesis is our metric systems engineering approach MeDIC, which addresses research question Q1. The details of the approach are outlined in section 1.5. To our knowledge this is the first time that a complete engineering approach for metric systems is defined. The two main parts of the engineering approach (answering research question Q2 and Q3 respectively) are additional major contributions of this thesis. We briefly provide an overview over these two parts in the next paragraphs. The details of each are found in part I and part II of this thesis.

The first major part of our contribution, addressing research question Q2, is a reference architecture for metric systems based on enterprise application integration, SOA and REST. The reference architecture specifically empowers flexibility and reuse of the measurement infrastructure components and a clear separation of the three main measurement infrastructure parts: measurement, calculation, and visualization. This enables fast, easy, and specific reaction to changes in the environment of the metric system; for example new stakeholders, new tools, or new visualization challenges. Additionally, the literature and our experience in the field indicated missing operation support in current measurement infrastructures. Existing metric research tools often include documentation about metrics and information needs. Therefore, we included these, a metric and information need focused documentation system and operation support, directly in the reference architecture to build metric systems that last for a long time. The details of the reference architecture can be found in part I of this thesis.

The second major part of our contribution, addressing research question Q3, is a process model for the development, construction, and operation of metric systems based

on best practices for metric requirements elicitation, up-to-date software development best practices, and established process models. Most existing metrics literature focuses on the requirements and early parts of the software development process. We included these best practices in our process model as well as specific activities to design, build, and operate reliable and flexible measurement infrastructures based on our reference architecture. Hence, we provide a process model that covers all parts of the software development process specifically for metric systems. The details of the process model can be found in part II of this thesis. Additionally, we provide a lot of details on specific activities, best practices, checklists, and document descriptions in the appendix of this thesis.

We do not provide a new notation, addressing research question Q4, in this thesis. Specific metric systems notations are required while specifying the metric system or in the documentation to provide an overview of the while system. Our experience, however, showed that existing UML and EAI-Pattern notations are sufficient for this. We provide a lot of examples for this in the definition of the reference architecture in part I and the field studies and tool description in part III of this thesis which also address research question Q5.

A side contribution is our formal approach to measurements provided in section 2.3. Our approach is based on established best practices and standards. We extended these to provide more flexibility as required by our process model and feeding our reference architecture. The formal approach provides a deep insight into the foundations of our interpretation of metric systems. Additionally, it enables conceptual termination proofs. It also provides formal requirements for reuse of metric system parts.

Metric reuse and a formal model for metric variability is another side contribution of this thesis. Flexibility and cost reduction are good drivers for reuse. Hence, we always included variability in our designs and formal approaches. Section 2.2 provides an inside into our metric reuse and variability concepts. We also include variability at key parts in the reference architecture and address reuse explicitly in the process model.

1.4. Research Field and Central Related Work

After we defined our research questions in the last section this section briefly introduces the core related articles and theses that contribute to our research field. One of the most related work to this thesis is the research on service-based measurement infrastructures by Martin Kunz during his time at Magdeburg working together with Rainer Dumke. Their work is aggregated in the PhD thesis of Martin Kunz: "Framework for Service-oriented Measurement Infrastructures" [Kun09]. The following first subsection describes the different parts of this thesis in detail. Another important work in our research field, especially the integration between GQM and measurement infrastructures, is the work on Software Project Control Centers performed by Jens Heidrich and others at Fraunhofer IESE. This work was embedded in the soft-pit project coordinated by Jürgen Münch. We provide a broad overview of their work in the second subsection. Our requirements from the field (section 1.1) are used to evaluate the two approaches and indicate missing parts. The last subsection concludes the research field and indicates central flaws and missing aspects in the two approaches.

1.4.1. Service-Oriented Measurement Infrastructures

Between 2005 and 2009 Martin Kunz et al. investigated service-based metrics and measurement infrastructures [KSDW06, Kun09, LDBK05, KMZB08]. The core of their approach is a service-oriented measurement process based on the ISO 15939. The process is supported by semantic measurement descriptions, a service repository, and a process called quality driven design. According to the ISO 15939 measurement process the measurement service is evaluated. The result of this evaluation is feed to the measurement process evaluation which influences the measurement process definition ([Kun09], p. 119).

The heart of the service-oriented measurement process is a service-oriented measurement database. The database feeds a measurement service for "performing measurement activities" ([Kun09], p. 134) such as: analysis, storage, collection, and measurement. The database itself has a very simple data schema. Metric, Measurement, and MeasurementResult form the core of the data schema. They are enriched by information about the quality model and associations to projects, organizations, and technical concepts (like classes) because they mainly consider static source code metrics for object oriented software ([Kun09], p. 160). The measurement service is divided into or[9] surrounded by several other services. Most importantly the data extraction service[10] which is feed with data by the resource view service, the process view service, and the product view service ([Kun09], p. 158). The measurement data is then feed to the visualization service[11] and the traffic light controller to visualize measurement values.

Another important aspect of the proposed infrastructure and process is quality based selection of services which is manifested in the $QuaD^2$ framework. The framework

[9] Unfortunately, the thesis is not very specific about this and most of the component and architecture diagrams contradict each other.

[10] The extraction service on page 158 is called import service on page 154 of [Kun09].

[11] The visualization service from page 158 is called presentation service on page 154 [Kun09].

requires a set of quality attributes for each service like restorability, memory utilization, and response time to select the services based on the requirements of the service consumer ([Kun09], p. 148 and [KMZB08]). The framework requires a quality model which is stored in a quality model database. The actual definition and tailoring of the quality model is supported by the experience factory.

Evaluation

We evaluate the approach by the means of our list of requirements from above. Similar to the requirements list we start with the process requirements and then move to the infrastructure requirements.

ReT-P1: Clearly defined development roles The process model proposed in the work covers important measurement stakeholders. From a development point of view, however, it only contains activities for the requirements and evaluation (maintenance) phase of the software development life cycle.

ReT-P2: Include developers and operators The process model is not focused on the construction of the measurement infrastructure, but focuses on the application of the infrastructure in an ISO 15939 oriented measurement process. Therefore, the important roles developer, architect and operator are missing.

ReT-P3: Continuous information need evaluation The strong ISO 15939 process orientation of the proposed process model implies continuous evaluation of the information need as continuous evaluation using a measurement experience base is the core of the ISO 15939 process model. Hence, it is also included in the proposed process.

ReT-P4: Encourage diversity The ISO 15939 process and accordingly this work focuses on individual measurements with specific phases for each measurement (plan measurement, perform measurement, and evaluate measurement). It also includes a diverse scope selection on the start which could be interpreted as a support for diverse metric systems and dashboards. Hence, this encourages diversity of the measurement infrastructure.

ReT-P5: Found on SE best practices The process model proposed is found on ISO 15939 and includes GQM. However, it is focused on the overall measurement process and not explicitly on the construction of the measurement infrastructure. It nicely reflects our experience from the field that software engineering best practices are not considered when working with measurement infrastructures and metric systems. As an example: the process model does not contain prototyping in the requirements (GQM) phase.

ReT-P6: Embrace flexibility by iterative and incremental development The process model does allow iterative and incremental development. But flexibility is not addressed explicitly. On the contrary, we believe the complex measurement definition and service selection is overly complex which hinders flexibility.

ReT-I1: Easy integration The service-based nature of the infrastructure supports extension. The measurement service can be extended by the means of extending the data extraction service. New visualizations can be added to the presentation service. However, we argue that the extension is not easy because the actual services need to be altered.

ReT-I2: Low latency The infrastructure stores measurement values in a central measurement database. The data is then consumed by the analyzation service and forwarded to the presentation service. Sadly the work does not provide additional information on the measurement latency. We, hence, conclude that latency was not a design issue and assume that it is not low explicitly.

ReT-I3: Separation The three important aspects system integration, metric calculation and visualization are clearly separated in the approach as already discussed with ReT-I1.

ReT-I4: Robustness Due to central storage of the data inside one measurement database and only single services for analysis, extraction, and presentation we conclude that the system is only partially robust. If one of these components fail all of the specific part will not work anymore. If the database is not available the whole system will stop working.

ReT-I5: Isolation The data extraction service just consumes data and is not integrated in other systems. The proposed measurement infrastructure is therefore isolated.

ReT-I6: No central data schema This is directly violated by only using a single measurement database.

ReT-I7: Evolution support Evolution of metrics, measurements, and visualizations is not explicitly addressed in the infrastructure. The approach also does not include mechanisms to avoid schema mapping and does not specifically address data evolution problems. However, the iterative process and separation of the central aspects into dedicated services supports it to a certain degree.

ReT-I8: Operation support Operation and development is not mentioned nor included in the infrastructure. It is also not included in the processes which again reflects our experience from the field that system operation is not something that is considered important when designing measurement infrastructures.

Discussion

The work from Kunz et al. contains a lot of important aspects and aligns the measurements with ISO 15939 and GQM. It also integrates service oriented ideas into the approach and provides a set of tools to evaluate the proposed ideas. From our experience, however, a lot of the ideas like service selection for the measurement services, the complex process feedback including the experience factory and huge measurement taxonomy are over engineered and cost-benefit ratio is not sufficient for an industrial application [VLJ13]. For example quality model driven selection of measurement services is a nice idea but typically there will only be one measurement service for a certain measurement.

Even though the work explicitly models the measurement process with some details ([Kun09], pp. 123-128) it completely lacks support for the actual software development and operation of measurement services. This also leads to a lot of violations to our requirements list from above. Operation and development support by the means of monitoring, logging, or configuration services is also missing in the infrastructure. These factors, however, are crucial to produce high quality distributed service oriented systems that last a long time [Jos07, BGR05]. Additionally, the work contains several inconsistencies, we mentioned a few of these in the previous sections, which often makes it hard to see the overall picture and follow the ideas.

1.4.2. Software Project Control Centers

Heidrich and Münch define Software Project Control Centers (SPCC) as follows (see [HM04], p. 4):

> [...] we define a SPCC as a means for process-accompanying interpretation and visualization of measurement data: It consists of (1) underlying techniques and methods to control software development projects and additional rules to select and combine them, (2) a logical architecture that clearly defines logical interfaces to its environment, and (3) a supporting tool that implements (parts of) the logical architecture.

They further propose an identification taxonomy which helps to classify SPCCs based on five different dimensions (purpose, technical, improvement, role, and tool). This taxonomy helps, as they used it, to analyze existing solutions. The dimensions also show a root source for the difficulties when specifying and constructing metric systems and measurement infrastructures: all these dimensions need to be addressed. Additionally, a solid measurement infrastructure and company wide metric systems often needs to address multiple items in each purpose category. Software Project Control Centers also enable control by the means of using innovative visualizations to gain project control [LHM+09]. The Specula tool/framework developed as part of this effort enables flexible visualizations be defining visualization catenas based on the underlying GQM analysis [HM08b, HM07, LHM+09].

The logical architecture supporting SPCCs contains three layers: the application layer, the functional layer, and the information layer. This architecture very much

reflects traditional three tier architectures for information systems [Fow02, Net14, Eck95a]. However, they specify a number of metric and control center specific components in these layers. In the bottom layer (information) they specify a reuse oriented component: Pool Management as well as two experience-bases: a project specific and an organizational one. The functional layer on top of this consists of components for typical BI-tasks: Customization, Data Processing, and Presentation. Additionally, this layer contains a Packaging-component. The focus of this component is abstraction and support tasks for the experience-bases. The application layer just contains a user-communication component as a front-end.

Heidrich and Münch also address the development of SPCCs by attaching the development of the SPCC to the software development process of the project that is supported by the SPCC. This is based on the "tailoring a measurement environment"' (TAME) software development model by Basili and Rombach [Rom91, BR88]. Later TAME was supported by the Specula approach developed as part of this effort [HM08b, HMW06a, Hei08].

The approach was evaluated by Ciolkowski et al. as part of the Soft-Pit research project [CHM+07, CHSR08, CHM08]. The evaluation shows that the approach is able to produce project control centers that allow to detect 80% of the listed plan derivations, faster risk identification than traditional approaches, and "people perceived the usefulness and ease of use of the Specula control center as positive"' [CHSR08]. The evaluation also provides a list of lessons learned. Among those they list the importance for a holistic approach towards software project quality control by the means of measurement. They also mention the importance of moving towards a learning organization by the means of using systematic project control mechanisms. This specifically provides benefits for small and medium sized companies by optimizing their development process. Last but not least they list scalability towards large projects as another success factor. Also in 2008 Ciolkowski et al. published a set of practical guidelines for the introduction of SPCCs in the industry [CHM08].

Evaluation

Again, we evaluate the work from Heidrich, Münch et al. based on the requirements defined in section 1.1.

ReT-P1: Clearly defined development roles Defined responsibilities are listed as an important practical guideline [CHM08]. However, this statement focuses on measurement responsibilities because the development process of the measurement system is not explicitly addressed in this work.

ReT-P2: Include developers and operators As mentioned before and similar to the work from Kunz et al., Heidrich et al. align their work with the ISO 15939 measurement process and GQM. Hence, they do not address the software development side of the measurement system and do not address developers and operators of the software control center.

ReT-P3: Continuous information need evaluation Again, similar to the previous work from Kunz et al., Heidrich et al. include an experience base to store measurement knowledge and continuously evaluate the measurement process.

ReT-P4: Encourage diversity Different scopes for different stakeholders are encouraged (requirement ReT-P4) as they state: "In general, an important success factor in the software engineering domain is that these solutions are customized to the specific goals, organizational characteristics and needs, as well as the concrete project environment." ([HM08b], p. 5). A customized solution is also listed as an important practical guideline [CHM08].

ReT-P5: Found on SE best practices The process and especially the tool support are based on current SE best practices. The Specula framework, for example, includes a model driven approach to define the measurement system by the means of visualization catenas [HM07]. However, other software engineering best practices are not included. For example: prototyping is not included in the requirements phase of the process to validate the GQM results.

ReT-P6: Embrace flexibility by iterative and incremental development Reuse and model-based initialization by the means of visualization catenas support flexibility. Incremental development is stated as one of the practical guidelines [CHM08]. However, flexibility was not states as a goal of the approach.

ReT-I1: Easy integration The service-based nature of the infrastructure supports extension. The measurement service can be extended by the means of extending the data extraction service. New visualizations can be added to the presentation service. However, we argue that the extension and hence integration of new data sources is not easy because the actual services need to be altered.

ReT-I2: Low latency The infrastructure stores measurement values in a central measurement database. The data is then consumed by the analyzation service and forwarded to the presentation service. Sadly the work does not provide additional information on the measurement latency. We, hence, conclude that latency was not a design issue and assume that it is not low explicitly.

ReT-I3: Separation The three important aspects system integration, metric calculation and visualization are clearly separated in the approach as already discussed with ReT-I1.

ReT-I4: Robustness Due to central storage of the data inside one measurement database and only single services for analysis, extraction, and presentation we conclude that the system is only partially robust following the arguments presented above.

ReT-I5: Isolation The data extraction service just consumes data and is not integrated in other systems. The measurement infrastructure is therefore isolated.

ReT-I6: No central data schema This is directly violated by only including a single measurement database and no mechanism to avoid schema mapping and data evolution problems.

ReT-I7: Evolution support Providing an easy-to-extend solution for SPCCs was the third central goal of the Specula approach ([Hei08], p. 8). Hence, the iterative process and separation of the central aspects into dedicated services supports evolution of metrics, measurements, and visualizations to a certain degree. However, the focus of the Specula approach is on cost-efficient setup and application of SPCCs rather than on producing maintainable measurement infrastructures (hypothesis H4, [Hei08], p. 9). Also, Extensibility, by the means of integrating new data processing techniques, and adaptability are listed as an important data processing requirement (PR3 and PR4, [Hei08], p. 30).

ReT-I8: Operation support Operation and development is not mentioned nor included in the infrastructure. It is also not included in the processes which again reflects our experience from the field that system operation is not something that is considered important when designing measurement infrastructures.

Discussion

The central and most important aspect of the work on SPCCs is the flexible and GQM-based combination of measurement data, control techniques, and visualizations. This is supported by an experience-base and aligned with goals and characteristics for the development project supported by the SPCC. Additionally, they align the SPCC development with the project development process. However, even though they provide a tool (Specula) and a reasonable logical architecture they did not provide a solid technical architecture. Additionally, their work also lacks a dedicated software development process model for the development of SPCCs. The interlinking between development project and metric system development is a feasible and good approach. However, this only addresses a very course grained level and misses specific activities for the development of metric systems like design, construction, and operation support. The lack of dedicated development and operation support might also be a reason for high additional cost overhead of 9% to 11%, which the approach produces for a development project of 6-15 developers [CHSR08].

1.4.3. Summary and Conclusion

Existing approaches are able to assist the overall measurement process defined in the ISO 15939. They specifically focus on the assistance of the measurement process by the means of experience-bases and taxonomies. Additionally, both approaches provide elaborate measurement support either by the means of a service-oriented measurement infrastructure or a full stack software project control center. Also, both approaches include a process component based on QGM to define measurement goals as a solid foundation for the utilized metrics.

The evaluation of both approaches based on our requirements specifically indicate the lack of dedicated software development support during the implementation of the metric systems. Especially requirement ReT-P5 and ReT-P2 as well as ReT-P1 to a large extend are not satisfied by both approaches. Focusing on the infrastructure aspect, both approaches utilize central data storage which violates requirement ReT-I6 and hinders ReT-I4 as well as ReT-I2. Additionally, (like many others) both approaches ignore operation of the measurement infrastructure and do not include specific tools to support it, which violates requirement ReT-I8.

Both approaches provide a solid framework to support metric initiatives and build service-oriented measurement infrastructures. However, both approaches lack the support for some of our core requirements. We assume that both approaches would also struggle with the challenges presented in section 1.1.3. Especially the challenges C1 and C2 are not addressed with both approaches. From this and the related work presented before we conclude that a dedicated metric systems engineering approach is still missing. Therefore, the following section presents the core of our approach, which is then supported by the foundations in the following chapter 2.

1.5. MeDIC - A Metric Systems Engineering Approach

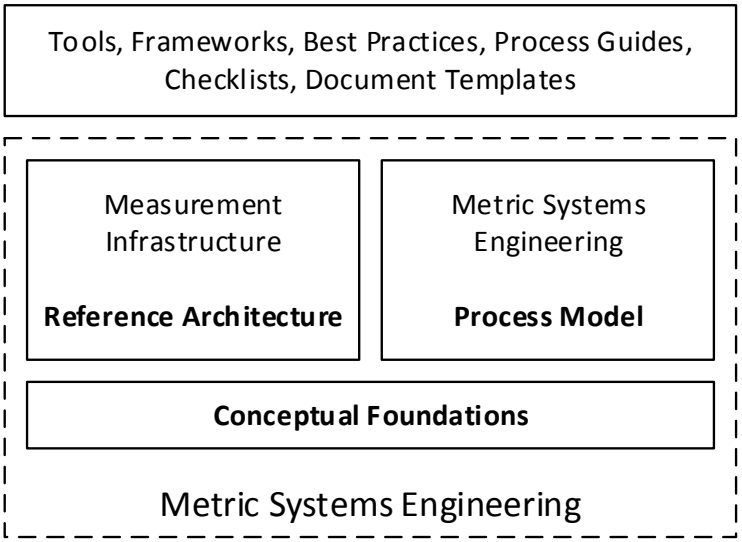


Figure 1.3.: The main parts of MeDIC

Reflecting the challenges presented in section 1.1.3 and as stated in the research questions the overall goal of this thesis is to provide an approach for

- engineering (development and operation)
- flexible,
- information need driven,
- and usable
- metric systems (concepts and implementation)

MeDIC addresses the core goal of our challenges from above to support the development and operation of successful metric systems in the long run. Hence, following common software engineering knowledge, the focus is not to be as cost efficient as possible during the initial implementation of the metric system but to produce professional solutions that are cheaper to maintain. Because over time maintenance costs will dwarf initial development costs.

The focus of this section is to provide an overview of our engineering approach: MeDIC. The main parts of MeDIC are depicted in figure 1.3. Most importantly, MeDIC contains a reference architecture for measurement infrastructures, presented in part II, as well as a process model to guide metric systems engineering activities, presented in part III. The basis of MeDIC are the conceptual foundations presented in chapter 2.

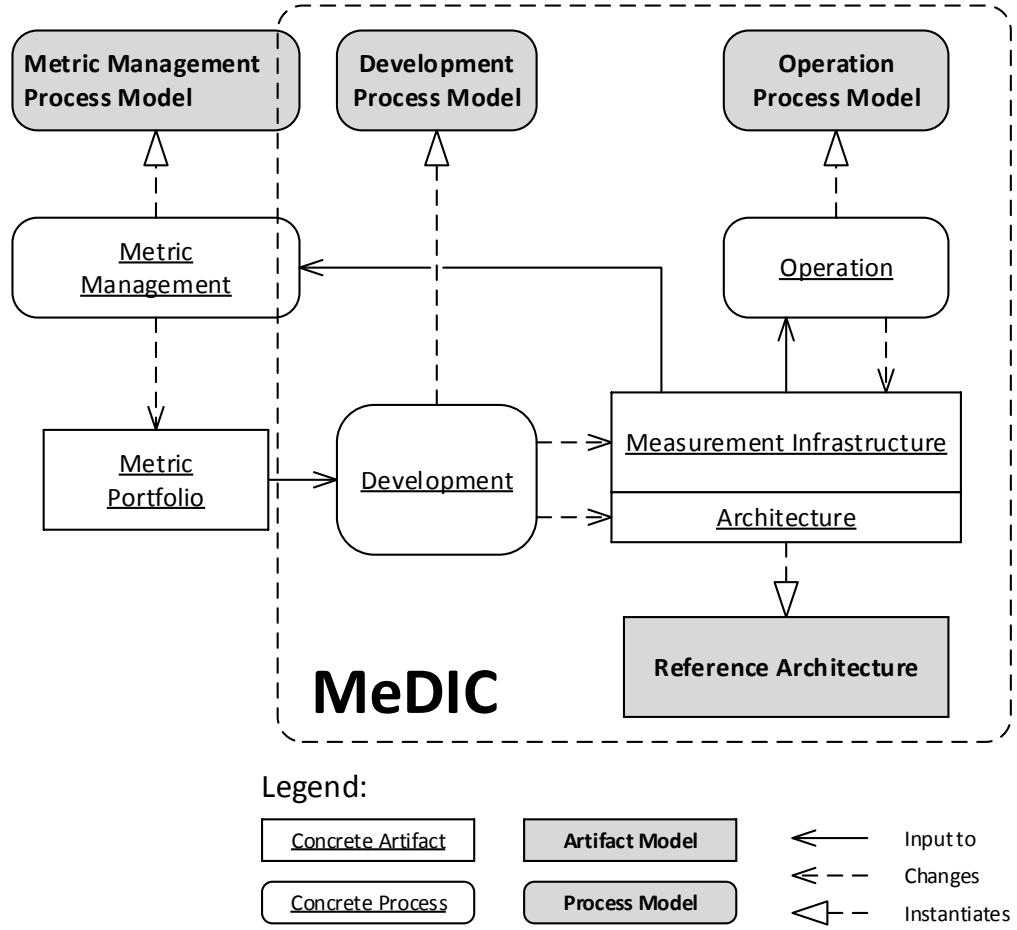


Figure 1.4.: MeDIC metric system engineering: Overview

Figure 1.4 provides an overview on the central processes, process models, and artifacts in metric systems engineering using MeDIC. It contrasts the situation depicted in figure 1.2 in section 1.1.3. Contrasting the chaotic and ad-hoc development of measurement infrastructures, MeDIC-based metric systems engineering includes a dedicated development process between the metric portfolio and the measurement infrastructure. The development process is based on the engineering process model included in MeDIC. Furthermore, the operation of the measurement infrastructure is also supported by operation processes. These are also based on the processes in the MeDIC engineering process model. The architecture of the measurement infrastructure is addressed specifically, in MeDIC base metric system engineering. The engineering of the architecture is also supported by the MeDIC measurement infrastructure reference architecture. As depicted in figure 1.4, MeDIC only scratches the metric management process and metric management process model.

To our knowledge and as discussed in the previous section existing approaches cover metric management quiet thoroughly. However, we some metric management aspects because the engineering process model and reference architecture need to rely on certain properties of metrics in the metric portfolio. The development process model also addresses the metrics requirements engineering phase, which is sometimes included in metric management process models as well.

The following subsections break down the different adjectives associated with the engineering approach for the two main parts. For each part we describe central design elements associated with the specific adjective. They also reflect our top level requirements from section 1.2.

1.5.1. Flexibility

Following our arguments for challenge C1 (section 1.1.3) metric systems will inevitably change over time because they need to reflect changes in their environment. MeDIC needs to address this with flexibility as required in the process requirements ReT-P3, ReT-P4, and especially ReT-P6 as well as the infrastructure requirements ReT-I1, ReT-I3, and specially ReT-I6 and ReT-I7.

Process Aspects

Most importantly, using the MeDIC process model stops big-bangs that introduce and changes huge parts of the metric system at once. Big-bangs cause problems because they change too much to fast. Hence, big-bangs will create a large overhead while developing the measurement infrastructure. Additionally it will raise problems with an overwhelming amount of stakeholders. Addressing these problems, the MeDIC process model follows an incremental development approach (requirement ReT-P6 and ReT-P5) that (should) result in small changes, which are easier to manage. Incremental development also allows to continuously adjust the metric system. Continuous evaluation is also included in the iterative nature of the process model (requirement ReT-P3 and ReT-P5). The information needs of the measurement customers are continuously evaluated to trigger new iterations of the development process. This reflects measurement best practices, which also require continuous evaluation of metrics typically supported by metric experience bases.

The MeDIC process model also addresses flexibility by including activities to define and include variability in metrics. As described in section 2.2.3 metric variability is a tool to reduce the amount of metrics by allowing metric definitions to contain variable aspects. Again, this features diverse metric systems (requirement ReT-P4). For example the metrics *number of change requests per month* and *number of change requests per week* are two different metrics. Classically, both of these metrics need to be defined, documented, and implemented. By using variability a metric expert is able to define a metric *number of change requests per time slot* in which *time slot* can either by *month* or *week*. This may sound trivial, but solid foundations and tool support for variability in metrics is still missing. Hence, we included metric variability ideas and concepts in MeDIC [Tav11, Mei11, Gre11, Röl13, Mäd12].

Infrastructure Aspects

Information needs and hence metrics and visualizations are constantly evolving to reflect changes in the environment of the metric system. Therefore, the measurement infrastructure needs to support flexibility and constant changes. Similar to the process aspects from above, MeDIC supports flexibility by avoiding a large monolithic measurement infrastructure (requirement ReT-I3). Most importantly MeDIC enforces measurement infrastructures with multiple heterogeneous databases with multiple different data schemata and redundancies (requirement ReT-I6 and ReT-I7). This avoids hard and dangerous data migration and schema migration activities. However, additional effort is needed to address problems related to redundancies and setup of the different databases.

Similar to above, metric customers should be able to tailor metrics and visualizations according to defined variabilities. This also needs to be supported by the measurement infrastructure. MeDIC reflects this by addressing variability in the metric calculation and visualization by specific parts of the measurement infrastructure.

Flexibility does come at a price, however. Most importantly, the measurement infrastructure needs to deal with the heterogeneity of data sources (additionally supporting requirement ReT-I1). We differentiate between different types of heterogeneity (see [Ste13], pp. 35-44). Most importantly: data type heterogeneity and data access heterogeneity. Both need to be addressed by different means. The reference architecture addresses data access heterogeneity at the lowest level: the data adapter level (see section 5.4 for further details) by the means of different data adaption patterns. Data type heterogeneity is addressed by the measurement data transport on the enterprise measurement data bus. Specifically by an object oriented and reuse focused type system on measurement messages. Further details on this are introduced in the conceptual foundations on metric system dynamics in section 2.3 and section 5.2 on data integration and measurement data transport.

1.5.2. Information Need Driven

Specifying clear information needs and metric goals is hard. Therefore, they need to be addressed early to find the real needs of the metric customers. Similar to other types of requirements it is important to get as much reviews and feedback in this early stage as possible (requirement ReT-P5). Hence, the MeDIC process model addresses the requirements phase iteratively and supports it with prototyping.

Information needs of metric customers make up the requirements for the measurement infrastructure. Hence, they need to drive the development and design decisions. They should also be referenced by the documentation and be accessible from the different tools. Thus, reflecting the requirements directly where the metric customers are interacting with the metric system.

Process Aspects

Existing approaches include experience-bases to support organizational learning on metrics definition. From our experience, however, this is very rarely used in industrial environments. Most metric definitions and metric documentation, if they are present at all, are plain text documents. Hence, metric documentation and heavy focus on metric definition early in the development process is a key part of the MeDIC process model. Additionally, information needs of the metric customers are the driving and directing force of the MeDIC process model. Ergo, the functional requirements of a metric infrastructure (the metric specifications) are rooted in specific information needs of metric customers. Reflecting the aspects from above, this information need focus helps to avoid big-bangs by incrementally focusing on a cohesive set of information needs at a time.

However, it is still hard to derive the *right* metrics and visualizations to satisfy the information need(s). Consequently, as mentioned above and following SE best practices, the MeDIC process model integrates prototyping activities after the initial information need gathering activities. Additionally, the MeDIC process model addresses and involves different roles involved with the metric system differently (requirement ReT-P1 and ReT-P2). As already mentioned several times: information needs will inevitably change over time. The MeDIC process model, hence, addresses this by continuously validating the metric system, for example by performing regular interviews with metric customers.

Infrastructure Aspects

All design decisions in the measurement infrastructure (according to the reference architecture elements) need to be justified by specific information needs of metric customers (or other metric related stakeholders). Colloquially summarized, the measurement infrastructure should be designed in way that only those things that are really needed should be build. A nice fitting saying goes: “Something is not good if you can add something. It is good if you can not take away from it anymore.”; we call this *information need focused*. This also interplays with separation of concerns when addressing the business view on a measurement infrastructure. MeDIC-based metric applications and infrastructure elements must reflect specific information need clusters. These clusters are also enforced by the coherent information need driven increments mentioned above.

Focusing more on the tool perspective of a metric system, dashboard tools should allow the metric customers to configure the dashboards on their level of abstraction. Their level are the information needs they have. Hence, dashboard tools should prefer configuration via information needs over direct access to metrics and visualizations. The reference architecture enables this with a specific reference architecture for flexible dashboard tools that can be configured by metric customers. Additionally, information needs should always be accessible (best: always visible) from the dashboard(s).

Operation of measurement infrastructures is often ignored. We followed the specific information needs from operators for measurement infrastructures in the design of our reference architecture (requirement ReT-P2 and ReT-I8). This lead to the integration of dedicated operation services and related tools in the reference architecture. Section 6.1 describes the details of this monitoring system.

1.5.3. Usable Metric Systems

The intention of metrics is often unclear to metric customers. The reasons for this are missing metric documentation, uncertainties in metric goals, and complex or innovative visualizations. Hence, the interpretation of the metric system becomes difficult; rendering the metric system almost useless. Consequently, the third goal of MeDIC is to produce usable metric systems.

Process Requirements

Most importantly, the MeDIC process model includes activities to document metrics and visualizations. This documentation contains the specifics of the metric like the measurement method or calculation formula. It also contains a description of the visualization as well as a list of possible metric customers. Following other approaches, the documentation also contains specific interpretation aids to guide the interpretation by the metric customers. Additionally, the MeDIC process model allows metric customers to analyze and interpret metric results on a collaborative basis. This can also be supported in the dashboard tools and collaboration results can be feed back to the metric experts.

Infrastructure Requirements

As mentioned before, information needs and interpretation aids should be included in the dashboards together with the visualizations. This eases the use for metric customers because the high level documentation (the information need) is directly located (and accessible) with the visualization.

Another important aspect to ensure usability is direct feedback for the metric customer. This, however, requires real time data processing in the measurement infrastructure (requirement ReT-I2). The reference architecture addresses this by favoring data push over data pull and specifically adding latency as an important design criterion.

As required in ReT-I4 the measurement infrastructure also needs to be robust to avoid complete application loss after a small local problem. The reference architecture is based on small (coherent) *independent* micro-services. If one of these services fail, the functionality provided by the service is no longer available but the rest of the application will still work. The MeDIC process model also contains specific activities to deal with service failures. These activities are also supported by specific services in the measurement infrastructure.

1.6. Summary

This chapter provided an introduction into metric systems engineering. Specifically, an introduction to our metric systems engineering approach MeDIC. We first presented metric systems engineering challenges that we faced during our field studies with industrial and research cooperation partners. From this we summarized three main challenges that a metric systems engineering approach needs to face. Most importantly we notices the need for flexibility and solid software engineering for the measurement infrastructure. We proceeded to formulate 14 top level requirements from the challenges. They are split into six requirements for the development process and eight requirements for the measurement infrastructure. We then discussed the requirements based on existing work in the literature. We concluded that all the requirements are valid and hence proceeded to formulate our research questions based on the requirements and challenges. Next, we presented an overview over our research field. We mainly discussed two research projects on *service-based measurement infrastructures* and *software project control centers*. The discussion was based on our top level requirements and concluded that a solid metric systems engineering approach is still missing. We then proceeded to present our metric systems engineering approach MeDIC. The goal of MeDIC is to provide and approach for flexible, information need driven, engineering (development and operation), of usable metric systems (concepts and implementation). These key adjectives for our engineering approach guided the overview of the two main parts of MeDIC: the reference architecture for measurement infrastructures and the metric systems engineering process model; thereby concluding the chapter.

The next chapter will formalize certain aspects which are only roughly described in this chapter. Most importantly it will define and decompose metric systems into their two parts: metric portfolio and measurement infrastructure. It will also provide a further and more formal introduction to our ideas for metric reuse and metric variability. Last but not least, the chapter features a formalism for metric system dynamics. This provides further requirements for the reference architecture for measurement infrastructure and certain activities in the metric systems engineering process model. It also enables to prove the termination of the calculation in a given metric system on a conceptual level.

2

Conceptual Foundations

The last chapter provided an introduction to metric systems engineering by presenting central engineering challenges and top level requirements. This chapter provides the foundations for our engineering approach MeDIC presented in the last chapter. We define the scope of our engineering approach by decomposing metric systems and defining central terms as well as their static and dynamic relations. These guide the definition of the reference architecture and the process model. Furthermore, the conceptual foundations introduce our approach to metric reuse, which is a key aspect in the process model and an important tool to reduce complexity in the metric portfolio and the measurement infrastructure. Finally, the conceptual foundations also introduces a formalism to model metric system dynamics. This allows to prove the termination of a given metric system (under certain conditions) on a conceptual level. The basis for the proofs also provides a number of constraints for the construction of certain elements in the measurement infrastructure. These constraints are later reflected in activities in the process model as well as the construction of the reference architecture. The definitions and examples also help to further understand our concepts in the metric portfolio and our idea for metric system dynamics.

2.1. Metric Portfolio

Before we dive into our ideas for metric reuse and the formalism in the end of the section, we first need to define central terms and dynamic relations between the different parts in the metric portfolio. The metric portfolio contains metrics and other central concepts for metric systems. From here on we use the term *metric definition* and *metric* synonymously because for us a metric is a definition of measurement. Thus, contrasting some definitions were metric also refers to "a set of figures or statistics that measure results" [1].

Metrics and indicators form the core of the metric portfolio. Their importance is also reflected in the fact that metric definitions are required to reach CMMI level 2 [Tea10]. Thorough metric definitions are a necessity for well-planed metric systems. Thus, this is also a crucial success factor for metric systems [HMO08, DLGP08b]. Hence, the topic of metric definition is addressed in a lot of research papers concerning metric documentation [PAFM04, dO03, DSZ06]. Most of these approaches are based on metric meta-models or on metric ontologies resulting in more formal definition rather than informal plain text. However, our experience shows that most of the metric definitions used in the industry (if they are used at all) are plain text documents. Sometimes these documents are on a more formal level by containing dedicated sections for specific attributes. For example the twelve steps to useful software metrics by Linda Westfall [Wes05], the required definitions for CMMI, or at least "goal", "question" and "metric" sections [Bas92].

The following subsections provide an overview of our understanding of the metric portfolio. This section defines the core static concepts and their dynamic behavior. After this we introduce our concept for metric variability which avoids an explosion of metric definitions by enabling dedicated reuse of metric definitions in section 2.2.

2.1.1. Metric Terminology

Figure 2.1 shows central terms and relations for the items in the metric portfolio using the UML class diagram notation. The diagram is separated into two parts. The left hand side contains the items that define the measurement and calculation of data inside a metric system. The right hand side contains the items that define the visualization and interpretation of those data. We first describe the items on the left and then continue to the right.

We use the term metric, similar to the majority of other approaches and the ISO 15939 standard [OM04, MGRP09, MJCH08, ISO07, CCP07, CET07], just as a naming container. The actual definition of the work performed by the metric is contained in the measurement approach of the metric. In accordance with our top goal for MeDIC, to keep everything as simple as possible, we define very little items in this model. This could be extended with a lot of additional information. But again we like to stick to the idea of: "Something is not good if you can add something. It is good if you can not take away from it anymore."

[1] See `http://www.oxforddictionaries.com/definition/english/metric` (in business).

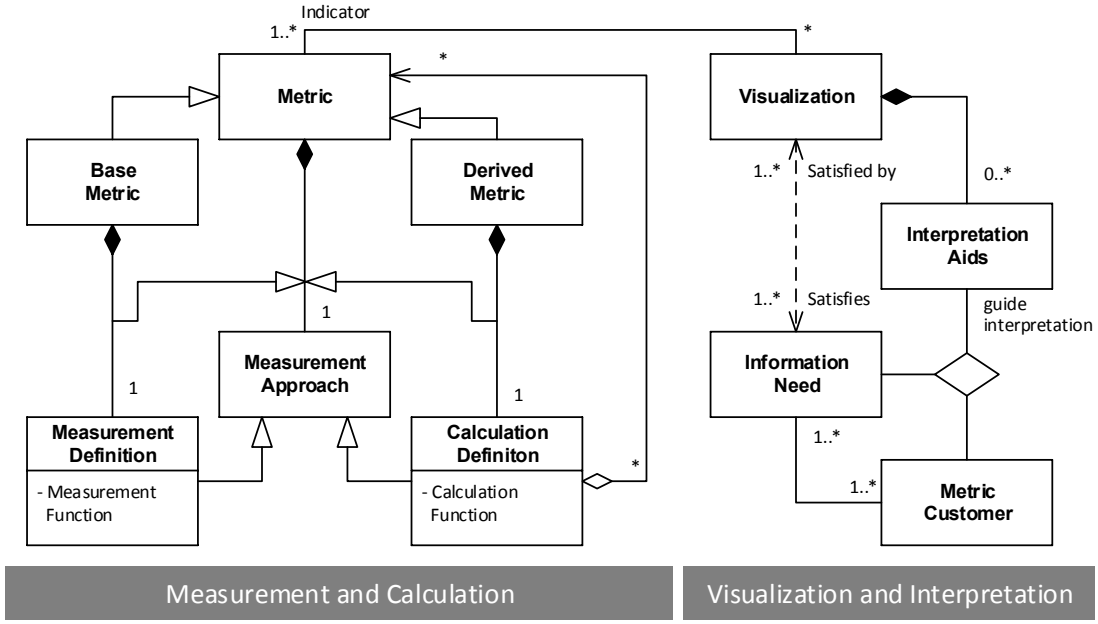


Figure 2.1.: Static relations between metric portfolio terms

In this thesis, again similar to all the other ontologies and metric meta models, we differentiate metrics into base metrics and derived metrics. The ISO 15939 defines base metrics as "a measure in terms of an attribute" [ISO07]. It further defines an attribute to be a "property or characteristic of an entity that can be distinguished quantitatively or qualitatively by human or automated means" [ISO07]. Derived measures are defined as "measure[s] that [are] defined as a function of two or more values of base [metrics]" [ISO07].

Following the ISO 15939, we need to differentiate the means of measurement – the measurement approach – for the two different types of metrics. We call the measurement approach for the base metrics measurement definition. It contains the measurement function which defines the measuring of data from a number of data providers. Furthermore, we call the measurement approach for derived metrics calculation definition. This contains the calculation function which defines how the data from other metrics need to be processed. Contrasting the ISO 15939 we do not limit the number of (base) metrics feeding the derived measures because in a practical application we also need to be able to define unary derived measures for example aggregation functions like "sum" or "avg". Additionally, we do not limit the source for derived metrics to base metrics.

The visualization definitions, as the name suggests, define the different visualizations in a metric system. The visualizations require data from the metrics. Again similar to the ISO 15939 we call this data indicators. Indicators can be simply measurement data or transformations of such data to suite a particular visualization. The visualization definitions can contain interpretation aids to guide metric customers. Each visualization satisfies information needs of metric customers.

2.1.2. Metrics System Dynamics and Measurement Data Flow

Following the literature and our definition from above a metric defines a function that specifies a measurement. Hence, a measurement is the application of a metric. However, this model does not define how measurements are interconnected. That is, how the output of one measurement is *feed* to another measurement. We call the model for the interconnection of measurement *metric system dynamics* because this is still on a conceptual level, hence *metric*, and defines the *dynamic* aspects of the interconnection between the applications of the metrics, the measurements. Hence, this provides a model for the measurement data flow in a metric system.

Metric system dynamics and the transition from metrics to something that executes measurements (the development process) is, as already mentioned several times, rarely addressed in the literature. However, we believe modeling metric system dynamics is as important as a good model for the static metric concepts.

Our model of the data flow in a metric system and hence our model for metric system dynamics is an extension of the data flow presented in the ISO 15939 standard [ISO07]. The ISO data flow is strictly tree oriented with data flowing from base measures to derived measures that feed other derived measures which then become indicators. The term definition for derived measures in the standard is even more restrictive by only allowing base measures to feed derived measures. This tree structure is stiff and hinders easy reuse of derived measures because they always require specific other measures feeding them. Our extension addresses this problem by opening the communication between metrics.

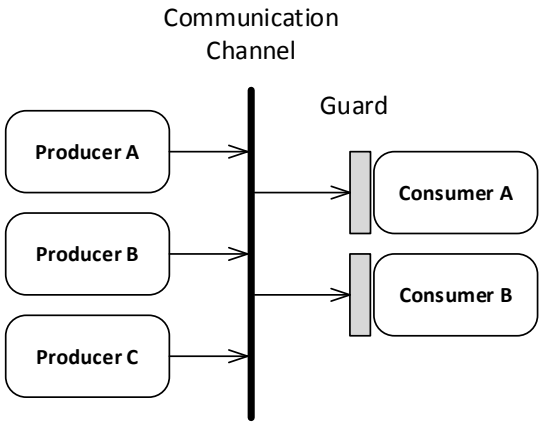


Figure 2.2.: Metric dynamics of a metric system by the means of measurement producers and measurement consumers

Rather then hard-wiring derived metrics and base metrics our idea is to model derived metrics as data consumers which consume suitable data (see section 2.3.6) on a communication channel from distributed data producers. Figure 2.2 provides a brief example of the dynamics between producers and consumers. The figure shows three measurement producers and two measurement consumers. The measurement data is

exchange on the communication channel which follows the publish/subscripe principle [HW03a]. Each measurement consumer is connected to the communication channel via a guard. This guard checks each measurement data on the communication channel before it is feed to the measurement consumer. This mechanism allows the consumers to only receive certain measurement data and ignore others.

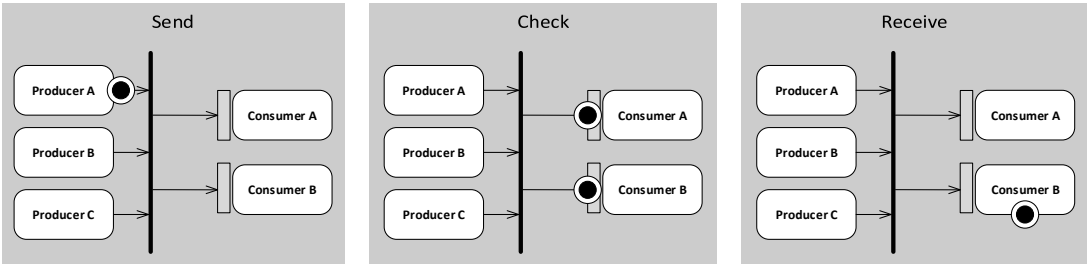


Figure 2.3.: Metric dynamics and measurement data flow example

Figure 2.3 provides an example of the exchange of a measurement data produced by Producer A. After production, the measurement data is transported to all guards of connected measurement consumers. Each guard checks whether the measurement data may pass or not. The guard from consumer A rejects the measurement data. The guard from consumer B accepts it. Hence, the measurement data is received by consumer B.

Figure 2.2 and 2.3 are simplifications. Real measurement infrastructure are build on multiple physical communication channels which each contains several logical communication channels. A formalism provides additional insights into the dynamics. Hence, we provides more details on the formal exchange of data between measurement producers and consumers in section 2.3. Additionally, this idea provides the foundations for the enterprise measurement data bus in the core of our reference architecture for measurement infrastructures presented in part II.

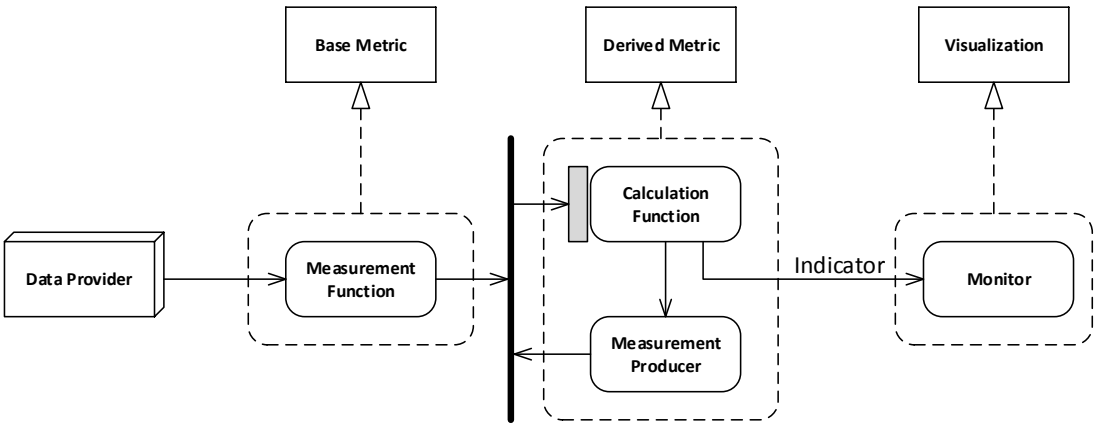


Figure 2.4.: Conceptual model of the data flow in a metric system

Figure 2.4 provides a brief overview of how we model the overall dynamics in metric

systems. The measurement data producers from above are either defined by base metrics or derived metrics. Base metrics utilize their measurement function to transport data from the data providers to the communication channel. Derived metrics consume measurement data which is then feed to their calculation function. Derived metrics utilize a number of producers (see section 2.3.7) to send derived measures to the communication channel. Similar to the ISO data flow and in accordance to our static structure from above, the derived metrics can also produce indicators which then satisfy the needs of metric customers.

2.1.3. Summary

This section presented the central terms and items in a metric portfolio as we see it. Our terminology is founded on common metric best practices and standards. Contrasting these, we kept the terminology very minimal. This minimalism helps to focus on the important aspects like the interplay between derived metrics and their inputs from other metrics. On the visualization side we also included a central stakeholder: The metric customer in the model. Hence, emphasizing the *information need driven* and *usability* aspect of our engineering approach.

Again contrasting a lot of popular articles, we also discussed metric system dynamic or measurement data flow. Measurement data producers and measurement data consumers create an implicit communication structure rather then an explicit one. Like all loosely coupled systems the price to pay for this flexibility is an increase of communication and structural complexity. However, this design also leads to a very flexible metric system and helps to reuse derived metrics. The next section will discuss the topic of metric reuse even further and extend it with our ideas for metric variability.

2.2. Metric Reuse

In the last section we defined a terminology for the definition of metrics and measurement data flow. However, defining metrics just for one project (in a multi project organization with a lot of similar projects) is costly and ineffective. Hence, it is wise to reuse metric experience (metric definitions, evaluations, and models) [DLGP08a] as all experience in software development can and should be reused [BR91]. Well-planned metric frameworks and reuse of existing metrics material is also mentioned as one key success factor for successful metric programs [HF97]. The work from Heidrich et al. as well as Kunz et al. covered in the central related work section (1.4) also heavily focus on reusing metric experience. Both approaches include experiences-bases to support iterative optimization of the metric portfolio.

Before introducing our approach to metric reuse, based on variability modeling similar to product line engineering, we investigate the different dimensions of metric reuse and discuss existing metric reuse approaches in the literature. The last part of this section contains a short overview of a formal definition for our metric variability model.

We investigated metric variability modeling in a variety of different theses from Tavizon, Meiliana, Mädler, and Röllig [Tav11, Mei11, Mäd12, Röl13] in the past. Consequently, this section summarizes a lot of the ideas that we investigated and provides a brief inside into the topic.

2.2.1. Metric Reuse Dimensions

Different aspects of metrics can be reused. Measurement tool reuse occurs very often because many organizations use the same LOC counters or even complete measurement tool suits like sonar qube [Son14]. Measurement processes like GQM [Bas92] are reused as well. The reuse of measurement values also increased during the last ten years. These (baseline) values are often used to enhance estimations. A popular example for this are the database and tools from the ISBSG [Lim, LWHS01]. Even though various measurement aspects are being reused metric specifications are very rarely reused. This ignores the fact that a database of metric specifications can spread metric knowledge across the organization and different projects (only 29% of the project managers and 24% of other practitioners know how measurement data was used in other measurement projects [PCN$^+$08]). Furthermore, the tailoring of modern metric based project management cockpits to fit the need of specific project roles [HMW06b] is a form of (implicit) reuse of metric specifications.

Although considerable research has been devoted to the modeling of metrics and metric frameworks, rather less attention has been paid to investigating how the results of this research (metric meta models, metric frameworks, and metric experience bases) can lead to a sound reuse concept for metrics and their specifications.

2.2.2. Metric Reuse in the Literature

Metric reuse is often implied or indicated but very rarely it is addressed. For example, the first three steps in the CAME Framework (Choice, Adjustment, Migration, Efficiency) by Dumke et al. indicate the benefits of an explicit modeling of metric variability [DK01]. However, the formal description of measurement and evaluation by Dumke and Schmietendorf [DSZ06] only mentions the importance of maintaining a metric experience base also included in the work of Kunz et al. [Kun09, KSDW06, KMZB08, LDBK05]. Later, Dumke et al. again imply the reuse of metrics because different usages and applications are modeled for measurement methods [DYAG09]. However, the concept of metric reuse is not covered in more detail. Hihn and Lewicki indicate a common set of standard metrics which is (re)used over several projects [HL11]. But no explicit tailoring of these metrics is mentioned nor is management or specification of variability. Starons and Medings pre configured wizards for the definition of metrics [SMN09] also implies metric reuse by automatically adding pre configured base measures. Again, reuse is not addressed by a sound concept but rather used pragmatically.

Most of nowadays model driven measurement approaches also imply metric reuse. For example as proposed by Clavel et al. [CET07] and extended by McQuillan and Power [MP06] by defining metrics based on UML concepts and OCL. Yet, this only allows the (pragmatic) reuse of complete metric definitions; only reusing fragments of metrics or the modeling of metric variability is not addressed. Reuse of existing metric components (like line charts, project plan structure and MS Project Import) is mentioned by Heidrich and Münch [HM08b]. But neither the reusable components nor their variability is explicitly modeled. Garcia et al. have proposed a model based environment for the integrated management of software measures [GSC+07]. They provide "generic metrics defined within the meta model scope" which according to the case study by Mora et al. on this environment homogenized the measurement process [MGRP09]. However, the variability of the metrics is again not reflected in the models (and in the meta model).

Reusable (sets of related) metrics are often represented by metric frameworks. According to Mendonsa and Basili these frameworks may also contain data collection mechanisms and information about data usage [MB00]. The framework implied by MIS-PyME – Software Measurement Maturity Model [DLGP08a, DLGP10, DLGP08b] suggests the reuse of existing measurement models of the organization, because "defining measurement programs for certain projects or products, ... will be costly, difficult to handle and of little worth for future developments" [DLGP08a]. Similarly, one of the goals of the INCAMI framework for (Web-based) metric documentation [OM04, dO03] is to "allows an organization to run different projects by making use of common measurement and evaluation mechanisms" [MO07]. But neither MIS-PyME nor INCAMI provide sound concepts for metric reuse.

Sets of reusable metrics could also be stored in an organizational wide metric experience base of a Learning Organization. As research by Krein et al. shows: providing a knowledge repository helps to push information back to the consumer [KWS+11]; in our case: supports reuse. Learning Organizations also avoid local optimization of

projects (and metrics) and focus on global optimization of the organization. The work of Althoff et al. indicates that learning organizations and avoidance of local optimizations reward reuse [ABT00]. Palza et al. describe specific metric experience bases which store the definition of and experience with specific metrics [PAFM04]. But, they also do not model metric reuse or metric variability. Metric experience bases are also a central aspect in the approach for software project control centers by Heidrich and Münch [HM04, HM08b, HMW06a]. Yet again, metric variability is not modeled explicitly.

2.2.3. Metric Reuse by Metric Variability

As addressed in section 2.2 we would like to reuse metric and visualization definitions from the metric portfolio. Unfortunately our investigations did not show a satisfiable approach for metric reuse in the existing literature. Our approach utilizes concepts from product line engineering to define reusable metrics. The details of this are listed in the following two sections. Furthermore, we define the formal foundation to our variability model.

Variability Modeling

Variability is an everlasting problem in software development and addressed in special areas like product line engineering [Kru02, Rom05]. We will focus on two possible solutions for dealing with variability: Parameterization (especially genericity), and variation points. Genericity, a special form of parameterized polymorphism, is a well known concept of programming languages like Java. Following Betrand Meyer, genericity "is a technique for defining elements that have more than one interpretation depending on parameters representing types" [Mic12]. In instantiating a concrete element from a generic one, the formal generic parameters need to be replaced by concrete types. Variation points are a concept from product line engineering. They are used to model the variability of a set of software products. Variation points are used to scope the system i.e. to determine what should be realized in a product line and what needs to be realized individually. Each variation point can specify a set of possible variations or allow all variations by remaining open. Additionally, constraints between variation points and variants can limit the configuration. A typical example for this is a variability model for cars in which the manual gear box, a variant of the gear box variation point, implies the selection of a specific clutch for the clutch variation point.

Metric Variability

We suggest applying a combination of genericity and variation points to realize variability in metric specifications [VLN12]. The "adaptation points" of reusable metric specifications are modeled as variation points. Of course, these variation points and the variants need to be clearly marked in the specification and need to be documented to ease and support tailoring. These variation points are the formal parameters of the reusable metric

specification. When reused, concrete values for all formal parameters need to be specified to derive a fully specified metric specification.

Metric variability allows a metric calculation, measurement, or visualization to define a number of variable aspects. The concept of metric variability is used to reduce the number of metrics defined in a real world environment by defining general metrics with variable aspects. For example a metric calculation for number of tickets could use slightly different inputs (trac tickets and jira tickets) for its calculation or different timings for the calculation output (daily, weekly, monthly). However, strictly speaking there is no such thing as Metric-Variability! Because every variable aspect in a metric will lead to a new metric. The example above really describes a number of different metrics for *number of trac tickets per day*, *number of trac tickets per week*, *number or track tickets per month*, *number of jira tickets per day* ....Since all these metrics are required in a real world scenario to answer slightly different information needs for slightly different situations this will inevitably lead to an explosion of metrics. Hence, from our experience with real world metric experts, metric customers, and architects for metric systems there is a need for variability in the metric specification.

2.2.4. Formal Foundation to Metric Variability

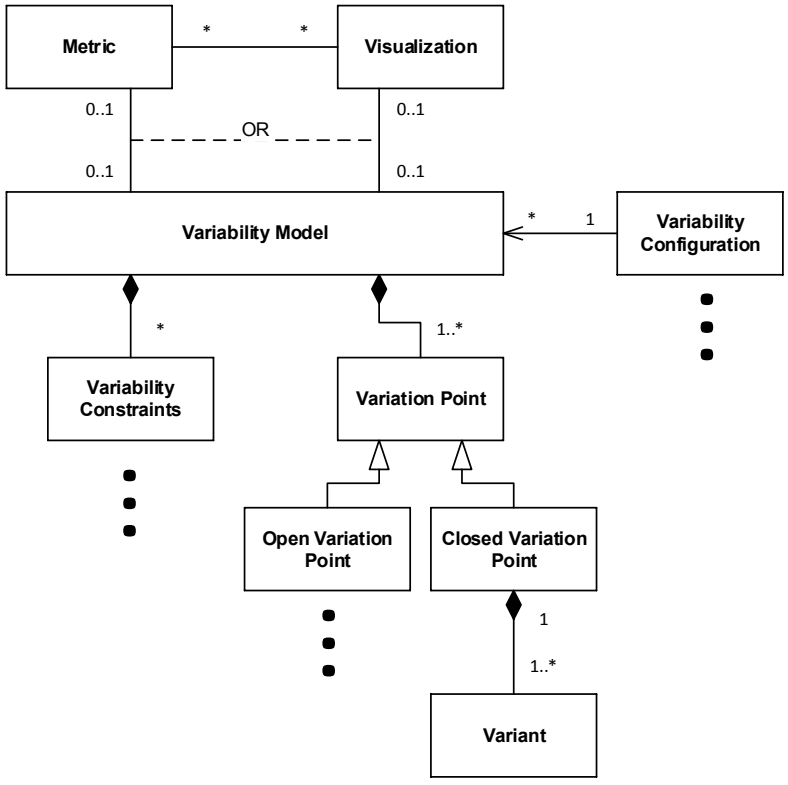


Figure 2.5.: Static view on central metric variability concepts

Variability (and its configuration) should not be applied arbitrarily. The boundaries for this should be defined by the metric experts. However, they need a formal framework to define these boundaries. Figure 2.5 shows a UML class diagram of the integration of a simple variability model attached to metric and visualization definitions. The variability model enables the definition of variation points and variability constraints which need to be defined in the context of the concrete metric system. The variability model distinguishes between open variation points and closed variation points. Open variation points allow an arbitrary configuration of a specific variation point contrasting closed variation points which only allow the selection of one (or more) of the given variants.

Each calculation function f_i provides a corresponding variability model VM_{f_i}. The variability model contains a set of variation points $VM_{f_i} = \{vp_1, \ldots, vp_n\}$ which can be either open or closed. Closed variation points simply contain the set of possible variants $vp_{closed} = \{var_{closed_1}, \ldots, var_{closed_k}\}$. Open variation points need to define the criteria for suitable variants for example as a (infinite) set or function. The actual variant of a given variability point vp in a config $conf$ can be accessed via the .-Notation: $conf.vp$.

2.2.5. Summary

This section presented our approach for metric reuse based on a flexible variability model. We realized, and our literature review shows, that a sound concept for metric reuse is still missing. Hence, we investigated the concept of metric variability from different approaches in past theses. These showed the benefits of a flexible variability model inspired by concepts from product line engineering. The core of the model is formed by open and closed variation points. Open variation points and variability constraints provide a flexible mechanism to tailor the variability model for specific applications. We also included a formalism for metric variability in the last subsection. The following section will continue these ideas by the means of a formalism for metric system dynamics from the initial section. This formalism will also include the variability model.

2.3. Formal Foundation to Metric System Dynamics

This section formalizes our abstract concepts for metric systems from sections 2.1.1 and 2.1.2. Our formalization to metric system dynamics should be suitable for the dynamics proposed before. Furthermore, the approach should contain a solid formal basis to measurement data and measurements. This should provide the formal basis to investigate termination of the calculation of measurements on a conceptual level. These investigations raise a number of important constraints for the construction of the (derived) metrics (see section 11.2.1 and appendix section B.2). These directly influence the construction of the reference architecture and activities in the engineering process model.

Figure 2.3 suggests a formalization to the exchange of measurements between measurement producers and measurement consumers similar to colored Petri nets. However, our formalism is not based on colored Petri nets because using colored Petri nets adds additional complexity to the formalism but does not reveal any additional insights over our more simplistic approach. Hence, our approach utilizes a combination of algebraic and functional formalisms to formalize metric system dynamics. However, our approach is compatible, and can be extended, to a formalism based on colored Petri nets; if needed.

We start this section with a brief discussion on some of the popular approaches to formalizing software measurements in subsection 2.3.1. After this, in subsection 2.3.2, we provide a rough overview over the central terms and relations in our approach. We then start with the formal definition of measurements and measurement data in subsection 2.3.4. The metric reuse concepts introduced in the last section requires a formalization of compatibility between measurements and measurement data, which is defined in subsection 2.3.5. subsection 2.3.6 defines satisfiability between measurement data and data types as the formal basis for the guards on measurement consumers introduced above in section 2.1.2. subsection 2.3.7 continuous these ideas by introducing a formal approach to measurement producers. These combine the measurements with metrics and measurement approaches from section 2.1.1 and also integrates the variability model and configurations from section 2.2.3 with variable derived metrics. The final subsection 2.3.8 then utilizes all these definitions to investigate calculation termination for a metric portfolio.

2.3.1. Related Work

Most of the existing formal approaches to measurement are approaches for model-based measurement. Some example for model-base approaches to measurement are the work by Mora et al. [MGRP09], by Staron et al. [SMN09, SM07, SMN09], and Lavazza et al. [LdBG08, Lav00, Lav05]. The models provide a good abstraction and solid foundation for the measurements. However, none of the papers that we investigated formalized the measurement process and metric system dynamics.

Formal foundations for software measurements emerged in the 1990s. Most noticeable work was contributed by Fenton and Zuse [Fen94, Zus91]. Other notable work was

contributed by Briand and Morasca [MB97, BEM96]. Their approaches map classic measurement theory to software measurement. However, maybe due to the trends of the time, they heavily focus on complexity measures and ignore process and project management metrics. Additionally, a core commonality for all their investigations is the view on metrics as a function that takes in some software entities and results a number. Fenton et al. themselves state ([Fen94] p. 203):

> It is popular in software engineering to use the word "metric" for any number extracted from a software entity.

A very rich overview on formal approaches to software measurements was given by Dumke, Schmittendorf, and Zuse in 2006 [DSZ06]. They categorize existing formal approaches into algebraic, axiomatic, functional, rule-based, structure-based, information-theoretic, and statistical approaches. Most of the examples for the different approaches again only see metrics as the measurement of a number. The algebraic and functional approaches, however, provide some flexibility in the representation of measurement data. Non of the examples provide a general abstraction of measurement data due to the theoretical nature of a lot of the papers. Additionally, non of the approaches reflect the ISO 15939 measurement model. Dumke et al. later integrated the ISO 15939 and declarative measurement approaches into the CAME approach [DBK+06]. CAME proposes a set of measurement principles, which are also provided on a formal level. Unfortunately, these principles again lack a formalization for measurement data. In 2009 Martin Kunz extended the CAME approach in his Phd thesis into a framework for service oriented measurement infrastructure [Kun09]. His approach uses web services and is heavily based on the ISO 15939. For more on his work also see our discussion in section 1.4.1.

This small subsection only scratched the surface of the body of knowledge on formal approaches to software measurement. However, all off the approaches that we investigated lack a formal definition for measurement data. Additionally, the formal abstractions presented hardly fit our loose coupling between metrics by the means of producers and consumers. Hence, we require a suitable formal abstraction for our metric system dynamics as defined above.

2.3.2. Overview

Figure 2.6 provides an overview over the central terms and relations of our formalism. The core of the formalism are measurements in the center. Measurements are produced and consumed by measurement producers and measurement consumers according to the dynamics provided in section 2.1.2. Measurements contain measurement data which are data records that are transported in the metric system. Measurements are also associated with an entity of measurement to identify the source of the measurement data. Measurement data and measurements are typed because a lot of the formal concepts (compatibility, satisfiability, and termination proof) are defined on their types. As defined in section 2.1.2 measurements are consumed by measurement consumers. Each consumer has an associated guard that decides whether a given measure should be consumed or

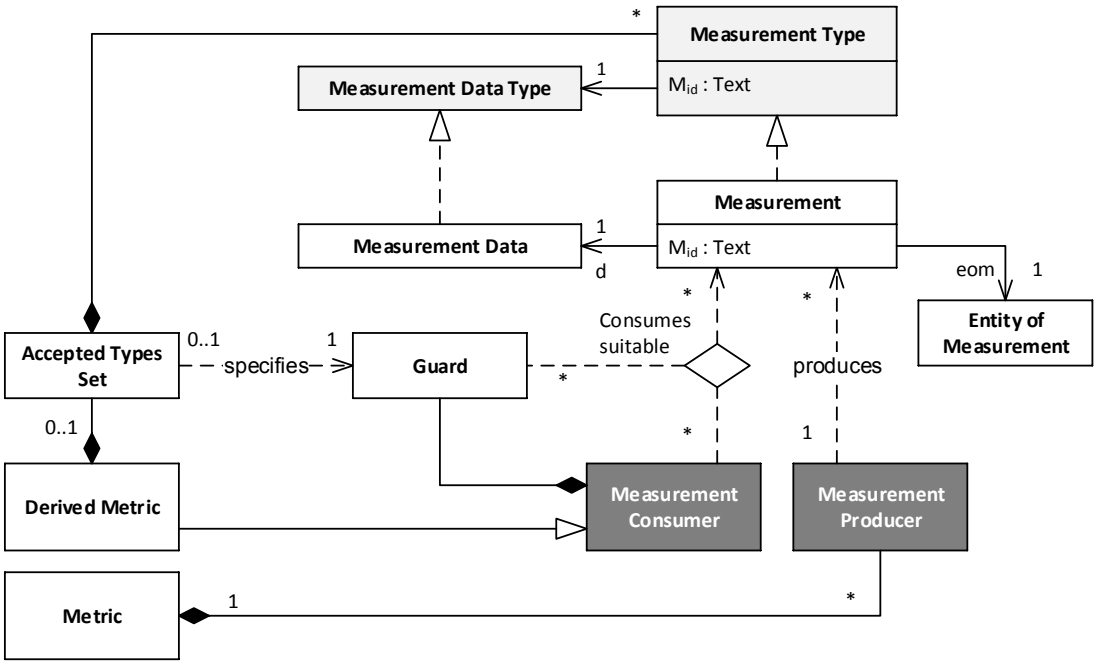


Figure 2.6.: Overview of the central parts to our formal foundations to metric system dynamics

not. For simplicity, we only consider special guards for derived metrics which are defined by a set of accepted measurement types as described in subsection 2.3.6 in the context of satisfiability.

We recommend to use figure 2.6 as a guide to remember the overall interactions between the parts while working through the details. Additionally, in the appendix in section A we provide a list of all the symbols used in the formalism to assist the reading of the formal notations.

We like to use our approach to prove calculation termination of a given metric system on a conceptual level. Therefore, we choose a combination of an algebraic and a functional measurement approach (see [DSZ06]). On the one hand, we use a set of algebraic definitions between measurements, measurement producers and measurement consumers together with formalized type requirements for measurement consumers to provide a tool to prove calculation termination. On the other hand, the calculation of the measurements use a functional approach by the means of calculation functions. The next section defines the basis for our formal approach: measurements and measurement data.

2.3.3. Preface

Throughout the formalism we often need to express that a formula is valid for all indexes in a certain range or that there exists a certain index in a given range. Consequently, we need to always define a set for this range in order to use the for-all or existence quantifier.

For example: $\forall\, i \in \{1, \ldots, n\}$. This wasts a lot of space and time to read. Therefore we define the following notation for the set:

$$\{1, \ldots, n\} = \overline{n}$$

Using this we can shorten the example from above to: $\forall\, i \in \overline{n}$.

2.3.4. Measurement Data and Measurements

Our approach to measures and metrics should not be limited to functions interpreting and resulting numerical values. Hence, we need a more flexible mechanism to express measurement data. Like most databases and programming languages we embrace the benefits of types. Hence, we also like our measurement data to be typed. We therefore first define the measurement data type before defining measurement data.

Measurement Data

Measurement data should store information that is important to or analyzed by calculation functions. However, we do not know before hand how the information is organized. This requires a very flexible approach to represent measurement data. In this thesis we, therefore, represent measurement data as sets of key-value-tuples. Additionally, we allow nesting of these records. Measurement data nesting simply allows for a value to again represent a set of key-value-tuples. Through data nesting we gain the flexibility and expressiveness necessary to represent arbitrary measurements.

Following our argumentation, we define a measurement data type T as a set of type components $t_1, \ldots, t_m$. Each of these type components t can either be a key for primitive values (represented as a string: key) or a tuple of a key and a nested type (key, T_{nested}) for nested data. In this case the keys can be interpreted as the measured attribute and the nested type as the type of the attribute if it is not primitive.

Consequently, measurement data d in this thesis is represented as a set of key-value pairs $d = \{(key_1, value_1), \ldots, (key_n, value_n)\} = \{kv_1, \ldots, kv_n\}$. According to our introduction, every $value$ can either be a (nested) set of key-value-tuples again or just a primitive value. However, we like to check and access the type of measurement data. Hence, we require a function $type()$ returning the type of measurement data d. To simplify the definition we first define a type function $type'()$ for the building blocks of measurement data: the key-value-tuples. Let $kv = (key, value)$ be a key-value-tuple of some measurement data. We define the function $type'()$ for a key-value pair as:

$$\begin{aligned} type &: key \times d \to t \\ type'(kv) &= \begin{cases} key & \text{if } value \text{ is primitive} \\ (key, T(value)) & \text{otherwise} \end{cases} \end{aligned}$$

Using this function we can now define the type function $type(d)$ for measurement data d as:

$$\begin{aligned} type &: d \to T \\ type(d) &= \{type'(kv_1), \ldots, type'(kv_n)\} \end{aligned}$$

This data representation is more powerful yet similar to an EAV (entity, attribute, value) data schema. Dinu and Nadkari investigated benefits and challenges for EAV data models in production systems [DN06]. They emphasize the flexibility and versatility of such data schemas and believe they are beneficial in situations where "Data are sparse, heterogeneous, have numerous attributes and new attributes are often needed." which is exactly the situation for data transported between metrics in a flexible metric system.

The data representation is also similar to the popular JSON data format and its superset YAML [EdNBK01, Bra14, Int13]. Our approach, however, does not differentiate between array and object data types for listed attributes. This differentiation is important for using JSON inside of javascript programs but the additional expressiveness is not required in our application of flexible data interchange between metrics. Hence, we only consider sets of records similar to JSON objects in which every data exists only once and do not consider a data structure that can contain duplicated entries similar to JSON arrays.

Measurements

The formalization of pure measurement data is not sufficient to formalize measurements. Measurements also require an identification of the object that was measured as well as a way to differentiate semantically different yet syntactical identical (type compatible) measurements[2]. Hence, a measurement in this thesis is defines as a tuple $\mathcal{M} = (d, M_{id}, eom)$ of the three properties: measurement data d, identification of the measurement M_{id}, and entity of measurement identification eom. The entity of measurement identifies the object that is the source of the data associated with the measurement. For example this could be the name or identifier of a project for an earned value measurements. Consequently, we define a measurement type as a tuple $\mathcal{T} = (T, M_{id})$ of a measurement data type T and a measurement identification M_{id}.

The measurement identification acts as additional key to the data to separate type compatible but semantically different measurement data from another. A simple example is a measurement of earned value (EV) and plan value (PV) of a project. Both are numerical typically represented by the following measurement data: $d_{EV} = \{(\text{value}, 123)\}$ and $d_{PV} = \{(\text{value}, 345)\}$. Unfortunately, these two measurement data have the same type: $T(d_{EV}) = \{value\} = T(d_{PV})$.

[2] As an alternative, the design of the formalism could simply define (force) that semantically different measurement data is also always syntactically different. Semantically different attributes, for example, would require different keys; making them syntactically different. However, for flexibility and reuse purposes later we like to be able to express (slightly) semantically different measurements with the same syntax.

Thus, the guard of a derived metric that calculates the schedule performance index (SPI) could not distinguish between the two. A possible solution would be to use different *value* keys. For example *ev.value* and *pv.value*. However, this would blow up the key structure for more complex measurement data. A better solution to overcome the problem is to introduce specific measurement identifiers and mark these two semantically different measurements with two different measurement identifiers. For example: $M_{id_{EV}} = \text{EV}$ and $M_{id_{PV}} = \text{PV}$. The guard can simply check the measurement identifier to distinguish between the different measurement data.

Similar to measurement data we can define a type function $type(\mathcal{M})$ for measurements that returns the type of the measurement. The type function returns a measurement type tuple that contains the type of the measurement data contained in the measurement and its measurement identifier. Let $\mathcal{M} = (d, M_{id}, eom)$ be a measurement. The type function $type(\mathcal{M})$ for measurements is defined as:

$$type : \mathcal{M} \rightarrow \mathcal{T} : type(\mathcal{M}) = (type(d), M_{id})$$

2.3.5. Compatibility

The initial sections of this chapter stated the importance of being able to reuse metrics. Systematic reuse of metrics, however, requires that metrics provide a clear definition of the measurement data that they produce and, in the case of derived metrics, which they require to consume. Systematic reuse also implies that we are able to provide generic solutions to typical questions. However, the measurements in a concrete metric system are typically very specific. Therefore, we need to be able to express compatibility and type compliance between generic and specific measurements and their types.

Typed data involves three different types of compatibility: compatibility between data types, compatibility between data records, and compatibility between a data record and a type[3]. For simplicity reasons, however, we only use one symbol and similar definitions for all three types of compatibility because the actual similarity type can be deduced out of the context.

On an abstract level compatibility is a directed relation *comp* between two entities a and b. The relation should be defined in a way that the tuple (a, b) is in the relation if a is semantical compatible to b. A simple example for a compatibility relation between measurement data and measurement data types is a type check relation *typeCheck* between a measurement data d and a measurement data type T. It contains the tuple (d, T) iff $type(d) = T$. Or short:

$$typeCheck = \{\, (d, T) \mid type(d) = T \,\}$$

[3]Compatibility between a data record and a data type is also called type conformance. That is: data a conforms to type T.

Measurement Data Compatibility

Agrawal and Wimmers propose an approach for record similarity related to inheritance relations [AW00]. Their approach to compatibility uses a preference function which forms a reflexive and transitive similarity relation on records (see definition 2.2 in [AW00]). A type test for such records was analyzed by Cohen and Watson already in 1990 and uses a similar abstraction [CHNP90].

We simplify their ideas and define compatibility between measurement data types based on the components of the measurement data types. Measurement data type compatibility is a directed relation $\prec$, as required above. Two types are compatible if for all keys in the right type there exist a compatible key on the left type. However, just checking the keys is insufficient because we also need to consider the nesting hierarchy in the types. Therefore, our definition will recursively check the nested types and stick to just checking the keys for primitives.

Before we look at complete types we define a help function $existsComp(t, T)$, between a type component t and a type T, which checks if there exists a compatible type component in T for t:

$$\begin{aligned} existsComp &: t \times T \rightarrow Boolean \\ existsComp(t,T) &= \begin{cases} \text{if } t = key\text{:} & \exists\, key' \in T :\ key = key' \\ \text{if } t = (key, T')\text{:} & \exists\, (key', T'') \in T :\ key = key' \wedge T'' \prec T' \end{cases} \end{aligned}$$

Using the *existsComp* help function from above we can now define compatibility between two types. Let $T_1 = \{t_{1,1}, \ldots, t_{1,n})\}$ and $T_2 = \{t_{2,1}, \ldots, t_{2,k}\}$ be measurement data types with their type components $t_{1,1}, \ldots, t_{2,k}$. We need to consider two different cases for the definition of type compatibility based on the number of type components for each type:

Case $n < k$**:** T_1 contains less type components than T_2. Therefore, T_2 represents a more complex type that T_1. Hence, T_1 can not be compatible to T_2 because not every key in each type component in T_2 can have a corresponding key in a type component in T_1. Therefore, $T_1 \not\prec T_2$.

Case $n >= k$**:** To check compatibility, we need to check if for each type component in T_2 there exists a compatible type component in T_1 using the help function from above:

$$T_1 \prec T_2 \quad \Leftrightarrow \quad \forall\, t_2 \in T_2 :\ existsComp(t_2, T_1) = true$$

Using compatibility between types we define compatibility between the measurement data. Let d_1 and d_2 be measurement data. d_1 is compatible to d_2 ($d_1 \prec d_2$) if their types are compatible:

$$d_1 \prec d_2 \quad \Leftrightarrow \quad type(d_1) \prec type(d_2)$$

Additionally, as stated in the introduction of this section we can also define compatibility between measurement data and measurement data types. Let d_1 be measurement data

with types $T_1 = type(d_1)$ and T_2 another measurement data type which T_1 is compatible to ($T_1 \prec T_2$). Then d_1 is compatible to all measurement data d_2 of type T_2:

$$d_1 \prec T_2 \quad \Leftrightarrow \quad d_1 \prec d_2 \quad \forall\, d_2 \text{ with } type(d_2) = T_2$$

Measurement Data Compatibility Example

We like to model measurement data from a ticket management system. The ticket system assigns priorities and severities to each ticket and it also allows to define custom properties. In this example we assume the custom properties "department" and "component". A suitable measurement data type to model such data would could be:

$$\begin{aligned} T \;=\; & \{\text{ticketId}, \text{severity}, \text{priority}, \\ & \quad (\text{custom properties}, \{\text{department}, \text{component}\}) \\ & \} \end{aligned}$$

An example for measurement data d according to T would be:

$$\begin{aligned} d \;=\; & \{ \\ & \quad (\text{ticketId}, 4711), \\ & \quad (\text{severity}, \text{medium}), \\ & \quad (\text{priority}, \text{medium}), \\ & \quad (\text{custom properties}, \{ \\ & \qquad (\text{department}, \text{development}), \\ & \qquad (\text{component}, \text{frontend controller}) \\ & \quad \}) \\ & \} \end{aligned}$$

Lets assume another ticket management system only contains ticketId, severity, and priority. A suitable type to model such data T' could be:

$$T' \;=\; \{\text{ticketId}, \text{severity}, \text{priority}\}$$

An example for such measurement data d' according to T' is:

$$\begin{aligned} d' \;=\; & \{ \\ & \quad (\text{ticketId}, 0815), \\ & \quad (\text{severity}, \text{low}) \\ & \quad (\text{priority}, \text{high}), \\ & \} \end{aligned}$$

According to the different definitions for compatibility it follows that:

$$d \prec T' \qquad d \prec d'$$
$$d' \not\prec T \qquad d' \not\prec d$$

Measurement Compatibility

We could simply extend the definition for measurement data to measurements with the compatibility definition from above for the measurement data of a measurement. Measurements, however, not only contain measurement data but also a measurement identifier. As a reminder: The identifier is used to distinguish between semantically different yet syntactical equivalent measurements. This semantical perspective needs to be considered by the compatibility relation between two measurements. Thus, the relation needs to take the identifier into account.

With this requirement we define measurement compatibility on a generic level as: Let $\mathcal{T}_a = (T_a, M_{id_a})$ and $\mathcal{T}_b = (T_b, M_{id_b})$ be two measurement types. Then $\mathcal{T}_a$ is compatible to $\mathcal{T}_b$ if its measurement data type T_a is compatible to T_b and its measurement identifier M_{id_a} is compatible to the measurement identifier M_{id_b}. Note that this definition requires a compatibility relation between the measurement identifiers. For an actual application of the formalism in this thesis, we need to define one.

$$\mathcal{T}_a \prec \mathcal{T}_b \quad \Leftrightarrow \quad T_a \prec T_b \wedge M_{id_a} \prec M_{id_b}$$

There are different ways to define a compatibility relation between measurement identifiers. Most of them require constraints on the measurement identifiers. For simplicity reasons, in this thesis we choose to use a namespace namespace concept on the measurement identifiers using "." as a separator between the namespace namespaces. Compatibility for measurement identifiers can now easily be defined using a prefix check. Hence, we define compatibility for measures as follows:

Let $M_{id_a} = a_1.a_2 \ldots a_{n-1}.a_n$ and $M_{id_b} = b_1.b_2 \ldots b_{m-1}.b_m$ be two measurement identifiers:

$$\begin{aligned} m < n \quad &\Rightarrow \quad M_{id_a} \not\prec M_{id_b} \\ m >= n \quad &\Rightarrow \quad M_{id_a} \prec M_{id_b} \quad \Leftrightarrow \quad \forall\, i \in \overline{n} \,:\, a_i = b_i \end{aligned}$$

For the example from above using EV and PV more suitable identifiers could be "projectmanagement.earnedvalue.ev" and "projectmanagement.earnedvalue.pv". The compatibility relation between the measurement identifiers M_{id_a} and M_{id_b} simply checks whether the string representation of M_{id_a} prefixes M_{id_b}. For example "projectmanagement.earnedvalue" would be compatible to "projectmanagement.earnedvalue.ev". This also makes sense semantically because a metric that is defined on all the different earned value data (pv, ac, ev, cpi, spi, cv, sv, ...) can certainly be calculated on ev data.

Again, the compatibility between measurement types can be extended to define

compatibility between two measurements. Let M_a and M_b be two measurements. Similar to measurement data, the measurements are compatible if their types are compatible:

$$M_a \prec M_b \quad \Leftrightarrow \quad type(M_a) \prec type(M_b)$$

2.3.6. Satisfiability

Our metric system dynamics use measurement sources and sinks as described above. The relations between the two, however, is not fixed. A measurement data consumer can specify what measurement (data) it is able to consume. We discussed the idea of guards above in section 2.1.2. In general a guard can be realized as a boolean function on measurements. If the function returns true the measurement is accepted by the guard.

Let $\mathcal{M}$ be a measurement. We define a guard function as:

$$guard : \mathcal{M} \rightarrow Boolean$$

We like to simplify this mechanic in a way that it closer reflects our ideas of measurement types and compatibility between measurements. We can use compatibility between measurements and measurement types to define a second order function that generates guard functions based on a measurement type:

Let $\mathcal{M}$ be a measurement, T a data type, M_{id_g} a measurement identifier and $\mathcal{T}$ a measurement type. We define a higher order function `guardGen` that generates guard functions for a given measurement type. This function can then be evaluated for a given measurement.

$$\texttt{guardGen} : \mathcal{T} \rightarrow \mathcal{M} \rightarrow Boolean : \quad \texttt{guardGen}(\mathcal{T})(M) = type(M) \prec \mathcal{T}$$

Measurement Satisfiability

From our experience, measurement consumers, defined by derived metrics, need to be able to accept multiple (different) types. Additionally, our later investigations require that we are able to express whether a given set of measurements is accepted by a guard function. Hence, we need to define a new concept that can be utilized by the guard functions. Contrasting satisfiability, however, this new concepts needs to work with sets of measurement types and set of measurements.

We call this concept "satisfiability" because the measurement set *satisfies* the criteria of the (implicit) guard function specified by the measurement type set.

Let $\mathbb{M} = \{\mathcal{M}_1, \ldots, \mathcal{M}_n\}$ be a set of measurements and $\mathbb{T} = \{\mathcal{T}_1, \ldots, \mathcal{T}_m\}$ be a set of measurement types like above. The measurement set $\mathbb{M}$ satisfies the measurement type set $\mathbb{T}$ iff the generated guard function for every measurement type from the measurement type set evaluates to true for a measurement of the measurement set:

$$\mathbb{M} \models \mathbb{T} \quad \Leftrightarrow \quad \forall \mathcal{T} \in \mathbb{T} \quad \exists \mathcal{M} \in \mathbb{M} : \quad \texttt{guardGen}(\mathcal{T})(\mathcal{M}) = true \tag{2.1}$$

Metric Reuse Implication

Let C be a measurement consumer defined by a derived metric that accepts the measurement type set $\mathbb{T}$. It consumes a measurement sets $\mathbb{M}$ that satisfies $\mathbb{T}$ ($\mathbb{M} \models \mathbb{T}$) in a certain metric system MS_A. Furthermore, let $\mathbb{M}'$ be another measurement set in another metric system MS_B. If every measurement $\mathcal{M}'$ in $\mathbb{M}'$ is compatible to a measurement $\mathcal{M}$ in $\mathbb{M}$ ($\mathcal{M}' \prec \mathcal{M}$) then $\mathbb{M}'$ also satisfies $\mathbb{T}$ ($\mathbb{M}' \models \mathbb{T}$). Hence, the measurement consumer C from metric system MS_A is reusable in the metric system MS_B consuming $\mathbb{M}'$. Thus, the set of accepted measurement types $\mathbb{T}$ can be interpreted as the *required interface* for its corresponding measurement consumer.

This drastically eases the design of reusable derived metric because they can be designed in a way to only specify the minimal data type required for their operation in their accepted types set. This derived metric can then be reused in a specific metric system that provides compatible measurements that satisfies the accepted types set. Additionally, using a similar argument it is also easy to design reusable base metrics. They only need to produce generic (minimal data) measurements which are easily consumed by different derived metrics which require more specific measurements in a broad variety of actual metric systems.

2.3.7. Measurement Producer

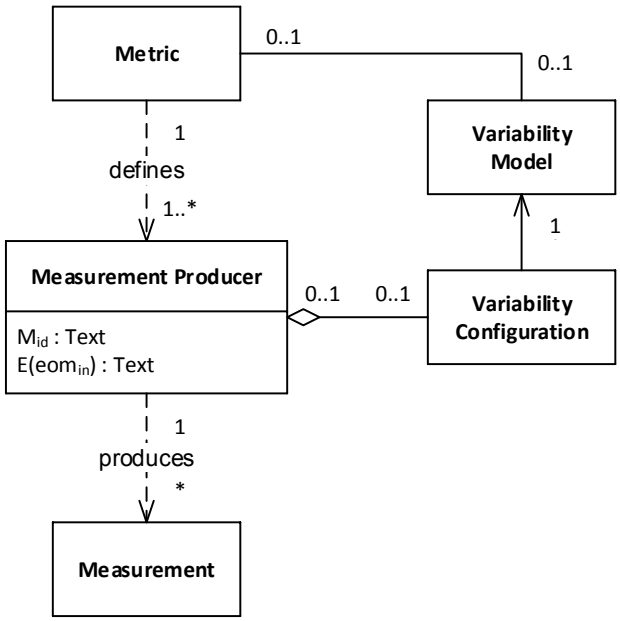


Figure 2.7.: Measurement producer connecting metric, measurement, and the variability model

Every metric defines the production of measurements. This measurement production connects our static view on metrics from section 2.1.1 with our formalism of measurements

presented in this section. Hence, we like to address the creation of measurements uniformly and specifically. We therefore include the measurement producers in the formalism. Figure 2.3.7 provides an overview over the important aspects associated with measurement producers. Metric on the top connect the static concepts from our terminology. Measurement on the bottom represents our dynamic formalism. Measurements are produced by the Measurement Producers in the middle, which glue metrics and their variability model together. Hence, this enables the specification of measurements produced by metrics with variability.

On the one hand, the mechanics of measurement producers without variability configuration for metrics without variability models is simple. They just execute the measurement or calculation function of the associated measurement approach of the metric. On the other hand, measurement producers for variable metrics, with associated variability model, need to be treated specially. The aim of the variability model was to reduce the number of metrics in the metric system. Our goal was not to specify each small variation of a metric. However, at this point a clear specification is required for the measurement production. The clear specification comes in the form of the variability configurations. Each variability configuration specifies the production of a specific measurement for a variable metric.

Variable base metrics are uncommon and very specific to the actual measurement approach. Hence, the remaining sections focuses on specific details of measurement sources for derived metrics with variability models.

Metric Product

Let a variable derived metric be associated with variability configurations $conf_1, \dots conf_h$ that suite the variability model of the metric. This derived metric, therefore, spawns the measurement producers $P_1, \dots, P_h$. Each measurement producer is associated with its specific variability configuration according to the index.

Measurement producers, as the name suggests, produce measurements. Let $\mathbb{M} = \{\mathcal{M}_1, \mathcal{M}_2, \dots\}$ be a set of measurements that is accepted by the guard function of the metric consumer with $\mathcal{M}_i = (d_i, M_{id_i}, eom_i)$ and $EOM = \{eom_1, eom_2, \dots\}$ the set of all entities of measurement from the measurements. The consumer then calculates the measurement data output by executing the calculation function f of the derived metric with the specific variability configuration $conf_i$ on the input data d. This calculation produces new measurement data. The measurement data outputs are then embedded in new measurements. The new measurements, however, each require a metric identifiers and an entity of measurements. The new metric identifier M_{id_i} is defined by each measurement producer P_i. They also define a function $E_{P_i} : EOM \to eom$ that calculates the new entity of measurements from the entities of measurement from the input measurements. Hence, the output of the measurement producer P_i is calculated as:

$$\begin{aligned} produce_{P_i} &: \mathbb{M} \to \mathcal{M} \\ produce_{P_i}(\mathbb{M}) &= \Big(f(d, conf_i),\ M_{id_i},\ E_{P_i}(EOM) \Big) \end{aligned}$$

The calculation of the entity of measurement provides important flexibility. Sometimes the measurement producer should return the same entity of measurement as the input measurement. For example the entity of measurement of the CPI is equal to the entity of measurement of the PV and EV in the measurement input. Sometimes the measurement producer needs to produce measurements with a different entity of measurement. For example if it calculates the spread between the maximum and minimum CPI of all the projects of the organization. This consumes the CPI values of the projects with their corresponding entity of measurement (project identifier). However, it needs to output the spread value with the entity of measurement of the organization because this metric is defined on the level of the organization.

We call the function that produces a set of all produced measurements for a specific measurement input $\mathbb{M}$ the production function of the derived metric $\mathfrak{M}$:

$$produce_{\mathfrak{M}} \quad : \quad \mathbb{M} \rightarrow \mathbb{M} \tag{2.2}$$

$$produce_{\mathfrak{M}}(\mathbb{M}) \quad = \quad \Big\{ produce_{P_1}(\mathbb{M}),\ \ldots,\ produce_{P_h}(\mathbb{M}) \Big\} \tag{2.3}$$

The output of $produce_{\mathfrak{M}}(\mathbb{M})$ (for a given set of input measurements) is called the product of the derived metric $\mathfrak{M}$.

Calculation Termination of a Single Derived Metric

The calculation functions and calculation results should be designed in a way that a calculation function for derived metrics does not process its own outputs (feedback) to avoid non terminating calculations. Cases in which the calculation terminates even if it does process its own output can easily be constructed[4]. However, if it does not process its own output then there is no feedback and the calculation undoubtedly terminates. Hence, it is sufficient to show that it does not process its own output to prove the termination of the calculation.

Let f be the calculation function of a (variable) derived metric $\mathfrak{M}$, $\mathbb{M}$ suitable input data, $P_1, \ldots, P_h$ the measurement producers for the derived metrics variability configurations $conf_1, \ldots conf_h$, and $\mathbb{T}$ the accepted types set that generates the guard function of the measurement consumer of the derived metric. Feedback in the derived metric is avoided iff no sub set from the output of the production function satisfies the accepted types set of the guard of the measurement consumer of the derived metric[5]:

$$\forall\, \mathbb{M}' \in \mathcal{P}(produce_{\mathfrak{M}}(\mathbb{M})) : \mathbb{M}' \not\models \mathbb{T}$$

The power set in equation 2.4 is calculation intensive. Hence, we like to further simplify it. We will prove, that transitive feedback is also avoided if only the product (without

[4] The derived metric requires a measurement A, produces a measurement B that is again consumed by it and produces measurement C. C, however, is not consumed. Hence, the calculation terminates even though it processes one of its own outputs.

[5] $\mathcal{P}(X)$ is the power set, the set of all sub sets, of X.

the power set) does not satisfy $\mathbb{T}$:

$$produce_{\mathfrak{M}}(\mathbb{M}) \not\models \mathbb{T} \tag{2.4}$$

The proof requires, that we show the following equivalence:

$$\forall\, \mathbb{M}' \in \mathcal{P}(produce_{\mathfrak{M}}(\mathbb{M})) : \mathbb{M}' \not\models \mathbb{T} \quad \Leftrightarrow \quad produce_{\mathfrak{M}}(\mathbb{M}) \not\models \mathbb{T}$$

Proof of the equivalence

We prove the equivalence by contraposition. We also invert the equivalence so the contradiction is more obvious. Hence, we need to prove the following two equations:

$$\begin{aligned} \nexists\, \mathbb{M}' \in \mathcal{P}(produce_{\mathfrak{M}}(\mathbb{M})) : \mathbb{M} \models \mathbb{T} \quad &\Leftarrow \quad produce_{\mathfrak{M}}(\mathbb{M}) \not\models \mathbb{T} \\ \nexists\, \mathbb{M}' \in \mathcal{P}(produce_{\mathfrak{M}}(\mathbb{M})) : \mathbb{M} \models \mathbb{T} \quad &\Rightarrow \quad produce_{\mathfrak{M}}(\mathbb{M}) \not\models \mathbb{T} \end{aligned}$$

We start with the first equation and prove the direction from right to left. Hence, we know that $produce_{\mathfrak{M}}(\mathbb{M}) \not\models \mathbb{T}$. Lets assume there exists a $\mathbb{M}' \in \mathcal{P}(produce_{\mathfrak{M}}(\mathbb{M}))$ that satisfies $\mathbb{T}$. From the definition of satisfiability in equation 2.1 follows:

$$\forall\, \mathcal{T} \in \mathbb{T} \quad \exists\, \mathcal{M} \in \mathbb{M}' : \quad \mathcal{M} \prec \mathcal{T}$$

The $\mathcal{M}$ on the right hand side is contained in $\mathbb{M}'$ which is a set from the power set $\mathcal{P}(produce_{\mathfrak{M}}(\mathbb{M}))$. If $\mathcal{M}$ is in $\mathbb{M}'$ which is contained in $\mathcal{P}(produce_{\mathfrak{M}}(\mathbb{M}))$ then, following the power set definition, $\mathcal{M}$ must also be contained in $produce_{\mathfrak{M}}(\mathbb{M})$. Hence, for every $\mathcal{T} \in \mathbb{T}$ there exits an $\mathcal{M} \in produce_{\mathfrak{M}}(\mathbb{M})$ that is compatible to $\mathcal{T}$. Therefore, $produce_{\mathfrak{M}}(\mathbb{M})$ also satisfies $\mathbb{T}$, which contradicts our initial criteria and we proved:

$$\forall\, \mathbb{M}' \in \mathcal{P}(produce_{\mathfrak{M}}(\mathbb{M})) : \mathbb{M} \not\models \mathbb{T} \quad \Leftarrow \quad produce_{\mathfrak{M}}(\mathbb{M}) \not\models \mathbb{T}$$

Now we need to prove the other direction to prove the equivalence. Hence, we know that $\nexists\, \mathbb{M}' \in \mathcal{P}(produce_{\mathfrak{M}}(\mathbb{M})) : \mathbb{M} \models \mathbb{T}$. Lets assume $produce_{\mathfrak{M}}(\mathbb{M})$ satisfies $\mathbb{T}$ then for each $\mathcal{T} \in \mathbb{T}$ there exists a compatible $\mathcal{M} \in produce_{\mathfrak{M}}(\mathbb{M})$. Again from the definition of the power set there exists at least one $\mathbb{M}' \in \mathcal{P}(produce_{\mathfrak{M}}(\mathbb{M}))$ that contain all these $\mathcal{M}$ for each $\mathcal{T}$. Hence, there exist a $\mathbb{M}' \in \mathcal{P}(produce_{\mathfrak{M}}(\mathbb{M}))$ that satisfies $\mathbb{T}$, which contradicts our initial criteria and we proved:

$$\forall\, \mathbb{M}' \in \mathcal{P}(produce_{\mathfrak{M}}(\mathbb{M})) : \mathbb{M} \not\models \mathbb{T} \quad \Rightarrow \quad produce_{\mathfrak{M}}(\mathbb{M}) \not\models \mathbb{T}$$

□

2.3.8. Calculation Termination

This section provides the formal basis to prove termination of calculation chains in a metric system on a conceptual level. Proving the termination of the calculations is important because infinite calculation inside an actual measurement infrastructure would massively drain resources and should, hence, be avoided. It is well known in computer science that proving termination in general is impossible. Our approach, however, is limited to the interaction between metrics and the investigations on the corresponding calculation chains. For these, as we show here, we can prove termination at least on a conceptual level.

Calculation Dependency

Using the definition for the product of a derived metric from above we define a calculation dependency between two derived metrics $\mathfrak{M}_a$ and $\mathfrak{M}_b$. Let $\mathbb{T}_a$ and $\mathbb{T}_b$ be the accepted type sets that generate the guard function for the measurement consumers of $\mathfrak{M}_a$ and $\mathfrak{M}_b$ respectively and $produce_{\mathfrak{M}_a}(\mathbb{M})$ the production function of the derived metric $\mathfrak{M}_a$ as defined above. We define the calculation dependency relation between $\mathfrak{M}_a$ and $\mathfrak{M}_b$ ($\mathfrak{M}_a \overrightarrow{\sim} \mathfrak{M}_b$ – read as: $\mathfrak{M}_a$ feeds $\mathfrak{M}_b$) iff one of the products of $\mathfrak{M}_a$ is consumed in the calculation of $\mathfrak{M}_b$.

$$\mathfrak{M}_a \overrightarrow{\sim} \mathfrak{M}_b \quad \Leftrightarrow \quad \exists\, \mathbb{M} : \quad \mathbb{M} \models \mathbb{T}_a \;\wedge\; \mathbb{M} \not\models \mathbb{T}_b \;\wedge\; \mathbb{M} \cup produce_{\mathfrak{M}_a}(\mathbb{M}) \models \mathbb{T}_b$$

Note that the input measurements $\mathbb{M}$ are required for the definition but the relation between $\mathfrak{M}_a$ and $\mathfrak{M}_b$ exists regardless of the input because the satisfaction relation only considers measurement types and not the actual data.

Calculation Chains

The derived metrics in a metric portfolio can form calculation chains[6] based on the calculation dependency between each other. A calculation chain has the form:

$$\mathfrak{M}_a \overrightarrow{\sim} \mathfrak{M}_b \overrightarrow{\sim} \ldots$$

For our further investigations it is important to identify specific metrics in a calculation chain. Hence, we formalize calculation chains using a vector notation. Let $\mathfrak{M}_a \overrightarrow{\sim} \mathfrak{M}_b \overrightarrow{\sim} \ldots$ be a calculation chain. We formalize this chain $\mathfrak{C}$ as the vector of the elements of the chain as follows:

$$\mathfrak{C} = (\mathfrak{M}_a, \mathfrak{M}_b, \ldots)$$

A derived metric in the chain can be accessed using the "." operator. For the example above $\mathfrak{C}.1 = \mathfrak{M}_a$, $\mathfrak{C}.2 = \mathfrak{M}_b$ and so forth. The length of the chain is calculated using the modulus operator $|\mathfrak{C}|$ by counting the number of derived metrics in the chain.

[6] More specifically they form calculation trees but we only consider the chains in these trees from the root to each leaf.

Termination

Following our arguments from above it is sufficient to show that each derived metric is only contained once in every chain (no transitive feedback) to show that all these calculation chains terminate[7]. To show that every derived metric is only contained once in every chain we need to show that for all chains all elements in the chain are distinct:

$$\forall\, \mathfrak{C}: \quad \forall\, i \in \overline{|\mathfrak{C}|}, \forall\, j \in \overline{|\mathfrak{C}|}: \quad i \neq j \;\Rightarrow\; \mathfrak{C}.i \neq \mathfrak{C}.j \tag{2.5}$$

Obviously, if equation 2.5 is satisfied in a metric portfolio equation 2.4 is also satisfied for all derived metrics in that metric portfolio. However, equation 2.4 is easier to check in the development process because it only requires to investigate one derived metric.

[7] Again, we could construct a similar example to above for a calculation chain that contains a derived metric multiple times and still terminates. However, proving and assuming that each derived metric is only contained once in every chain is very easy and simplifies the termination investigation.

2.4. Summary

This chapter presented the foundations to our metric systems engineering approach MeDIC. We presented our metric terminology based on existing approaches for metric documentation, metric meta models, and metric taxonomies. The terminology addresses the two parts *measurement and calculation* and *visualization and interpretation*. Our model for metric system dynamics on top of the taxonomy uses a flexible system of measurement sources and measurement sinks for the exchange of measurements. This obviously supports *flexibility* of our engineering approach. Additionally, the inclusion of the metric customer in the taxonomy supports the *information need driven* and *usability* aspects.

Our variability-based approach to metric reuse avoids an explosion of metric definitions. The approach adds a variability model to the metrics. This variability model enables the definition of variation points in the definition of metrics. This approach was supported by an elaborated analysis of related work in the area of metric reuse and a formal definition of the variability model.

The second half of the chapter was dedicated to our formal approach to metric system dynamics. This utilizes a combination of a functional and algebraic approach to formalize measurements in metric systems. The approach provides additional requirements to the design of derived metrics and solidifies our ideas for reuse of metrics. The approach also provides a framework to prove the termination of the calculation of a given metric portfolio under specific circumstances.

The following part utilizes these foundations to define our reference architecture for measurement infrastructures. After an overview to the static and dynamic aspects to the reference architecture, which instantiate the ideas from this chapter, we continue the formal approach from this chapter. The resulting formalism on the specific reference architecture items enables termination proofs and the calculation of an outer calculation hull on real architectures for measurement infrastructures.

Part II.

Reference Architecture

for Enterprise Measurement Infrastructures

3

Introduction, Requirements and Foundations

In the last two chapters we discussed the main challenges to metric systems engineering, provided a broad overview of our approach MeDIC, and presented the foundations to MeDIC. This part presents one of our main contributions to metric systems engineering: our reference architecture for measurement infrastructures.

A solid and sound architecture is key to the long time success of a software system [Per00, DdOdlP98]. However, its design is difficult and requires a lot of experience. Reference architectures address these issues by providing a framework for the design of a concrete architecture. Our reference architecture is a model for the architecture of measurement infrastructures. It contains elements that provide a suitable abstraction for the elements required in the architectures for actual measurement infrastructures. Additionally, we also provide micro reference architectures for each of these elements, which guide designing the actual services for an actual measurement infrastructure. With our reference architecture we address the following architectural views (see the 4+1 architectural view model from Philippe Kruchten for further details [Kru95]):

Logical Architecture Provides a higher level abstraction based on the logical concepts. In our case the logical reference architecture provides a framework for the different logical components of a measurement infrastructure. The logical reference architecture also assist the decomposition of a measurement infrastructure by the means of (logical) metric applications.

Physical System View The physical system view provides a view on the systems in their deployment state on physical nodes in a network. In our reference architecture we use the physical system view to classify the systems in the core of the measurement infrastructure.

Technical Architecture The technical architecture refers to the design and organization of the technical concepts. It contains services and layers which tackle different measurement challenges with specific solutions. Therefore, the technical reference architecture supports the design activities of our process model. In the 4+1 model this is called the "development view". We believe, however, this name is for this case to closely related to the development activity of the process view

In the reference architecture we do not provide the process view of the 4+1 model. This is addressed by our metric systems engineering process model in part III of this thesis.

We call measurement infrastructures based on the MeDIC reference architecture *Enterprise Measurement Infrastructures* (EMIs) to differentiate them from ad-hoc measurement infrastructures. An enterprise measurement infrastructure therefore is an instance of our reference architecture.

We first investigate the low level requirements for the reference architecture in section 3.1 based on the top level requirements from section 1.2. After this we briefly investigate alternatives and related work in section 3.3. The reference architecture definition then starts with our logical reference architecture and the physical system view in chapter 4. In there we provide our view on the logical architecture for metric systems. This guides the decomposition of a metric system and provides a high level overview of the different parts of a metric system. Furthermore, we provide a classification for the physical systems in an EMI. Chapter 5 is the core of this part. It provides a detailed overview over the technical reference architecture. After an overview over the layers and dedicated services we zoom into each layer and provide dedicated discussions and micro reference architectures for the specific services. The following chapter 6 provides overviews of the requirements, logical, and technical reference architecture for the most crucial operation systems; the most important once being the monitoring system and the logging system. These two drastically ease the development and operation of a measurement infrastructure. We close the part with a formal-basis to our reference architecture in chapter 7. This contains a formalism for the service operation states and, more importantly, a full formalism for our technical reference architecture. This extends our formalism from section 2.3 with formalisms for our technical concepts e.g. measurement messages, data adapters, and metric kernels. This provides the basis for investigating the technical adaption (implementation) of the metric concepts and provides a means to calculate the "reach" of an EMI based on the raw data provided by the data provider.

3.1. Design Foundations and Reference Architecture Requirements

Before we introduce our reference architecture we refine our top level requirements from section 1.2. This refinement also includes a lot of important design decisions for our reference architecture. However, this section does not cover specific requirements for the tools that are accessed by the metric customers of an actual measurement infrastructure! The reason for this is that a lot of different goals need to be considered when defining the tool requirements and a valid selection of important goals is only possible when dealing with an actual metric system with actual stakeholders. The stakeholders (specifically metric customers and metric experts) need to decide which goals need to be addressed and which are not. Therefore, they need to be addressed when designing the specific architecture for a measurement infrastructure for an actual metric system.

3.1.1. Polylithic Micro Service-based Measurement Infrastructures

Most importantly our reference architecture should address challenge C1 from section 1.1.3; it need to support flexible measurement infrastructures! Consequently the reference architecture needs to address the top level requirements ReT-I1, ReT-I3, ReT-I6, and ReT-I7. Our investigation in the beginning showed that a lot of the problems are due to monolithic infrastructures and large monolithic tools. Hence, our solution for this is to enforce a polylithic over a monolithic design of the infrastructure. Following the Unix philosophy we like our reference architecture to follow the rule of modularity, rule of clarity, rule of composition, rule of separation, rule of simplicity ([HT99], p. 76). We believe this leads to a flexible and expandable measurement infrastructure and helps to flexibly evolve the different parts of the measurement infrastructure as required. An additional benefit is that this does not force a specific technology on all the different services of the measurement infrastructure and the architect does not need to worry about unifying the technology of the different services.

Recently this design principle got known as *microservice architectural style*. The core idea of microservices is to build a large system based on small cohesive and independent services. James Hughes defines microservices as [Hug13]:

> Micro Service Architecture is an architectural concept that aims to decouple a solution by decomposing functionality into discrete services. Think of it as applying many of the principles of SOLID at an architectural level, instead of classes you've got services.

SOLID is an acronym for the principles of: Single responsibility, Open-closed, Liskov substitution, Interface segregation, and Dependency inversion [Met09]. We believe that adhering to these core principles, while designing services, leads to solutions that are easier and cheaper to maintain and hence evolve more easily. Martin Fowler advertises microservices over the past years. He and James Lewis define microservices as [FL14]:

> In short, the microservice architectural style [1] is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

Especially the last part fits nicely to our requirement ReT-I6 for no central data schema and individual data base choice per service. A microservice oriented architecture, however, goes beyond free (and therefore optimal) choice of databases per service. The architect is also free to choose the optimal implementation technology for each service. If for example the solution requires transactions and fast responses with high integrity constraints on the data then for example a Java enterprise solution using a relation database is a good choice. On the contrary if the data is graph oriented with a lot of complexity in the interconnection of the data and the solution should be very light weight and easy to deploy then a javascript based node.js backend using a graph database forms a feasible solution. In a microservice oriented architecture these two services could happily coexist in the same system to serve a bigger purpose. These examples, however, also show that it is crucial to stick to the single responsibility principle and aim for very cohesive and small services.

3.1.2. Specific Requirements

The single responsibility principle is directly related to the design principle of separation of concerns. Our top level requirement ReT-I3 already forces a clear separation of the core tasks of a metric system: Measurement, Calculation, and Visualization. Because we chose microservices as the design metaphor we also need to consider a fourth task: service integration. This section drills down into the fine detailed requirements for each task. In the following section we choose the word *part* for the high level items in the reference architecture because we do not want to emphasize a specific solution in this requirements section. *Layer*, *component*, or *service* would already suggest a specific solution. The smaller things inside the parts are, thus, called *items*.

Requirements for Measurement

The measurement part of a measurement infrastructure of a metric system connects the measurement infrastructure to the data providers (see system overview in section 4.2). Hence, the measurement part is the first one that needs to deal with the heterogeneity of the measurement data and needs to provide a solution that fits requirement ReT-I2 (fast and up-to-date data recognition). We also need to make sure to obey to requirement ReT-I5 to isolate the measurement part from the data provider.

ReD-IM1: Deal with heterogeneous data access The measurement data from the data providers can not be accessed uniformly. Hence, the measurement part of the reference architecture needs to deal with this heterogeneous data access in a way that it provides multiple solutions for different (typical) measurement scenarios. Most importantly, the different solutions in the measurement part of the reference architecture need to define and document the different heterogeneity scenarios.

ReD-IM2: Enable real-time data The measurement part of the reference architecture needs to provide a solution that allows real-time data access. We realized, however, that this is not always possible and is closely related to the first requirement for heterogeneous data access. Therefore, again, multiple solutions are required to deal with different measurement scenarios.

ReD-IM3: Isolate the data providers The measurement solution needs to be isolated from the data provider in a way that a failure in the measurement part does not result in failures or problems in the data provider. Furthermore, the measurement part of the reference architecture needs to define mechanisms to deal with failures and faults of the measurement and of the measurement infrastructure.

Requirements for Integration

The integration part of the measurement infrastructure integrates the measurement and the calculation part. This part is specifically important to our solution because we choose the microservice architecture style as a basis for our reference architecture for polylithical solutions. This leads to a lot of different services that need to be integrated. A monolithical solution also requires the integration between the different parts but, due to higher coupling between the parts, it is not as important. Furthermore, the integration part needs to support the top level requirement ReT-I4 that requires a robust solution.

ReD-II1: Deal with heterogeneous data types We already discussed the different types of heterogeneity in the measurement part (also, see [Ste13]). The measurement part deals with the heterogeneity of the measurement data access. The measurement data itself, however, is also heterogeneous. Hence, the integration part of the reference architecture needs to provide a solution to unify heterogeneous measurement data. Additionally, it needs to support the implementation of the satisfiability and compatibility relations defined in the formalism in sections 2.3.6 and 2.3.5.

ReD-II2: Transport measurement data The core requirement of the integration part is, as mentioned above, the transport of measurements from the measurement part to the calculation part and inside the calculation part. Also as mentioned in the last requirement this needs to obey the requirements from our formalism in section 2.3.

ReD-II3: Obey the criteria of Hohpe and Wolf Hohpe and Wolf define a set of criteria that an integration solution should obey [HW03a].

ReD-II4: Isolate the different parts Following the top level requirement ReT-I4 the integration part of the reference architecture needs to enable the isolation of different parts and services from each other. Failures or faults in one part or service must not result in an overall failure or fault of the whole measurement infrastructure.

Requirements for Calculation

The calculation part is the core of a measurement infrastructure. This is where all the metrics are calculated. The visualization part, therefore, connects to it to get its data to satisfy the information needs of the metric customers. The most important top level requirements to this part are ReT-I2, ReT-I6, and ReT-I7.

ReD-IC1: Calculation is independent from measurement Following our discussion in the formalism (specifically sections 2.3.6 and 2.3.5 about satisfiability and compatibility) the measurement part and the calculation part are only coupled via the measurement types. From the perspective of the calculation part the accepted measurement types of derived metrics act as required interfaces. These are satisfied by measurements produced either by other calculations or by the measurement part. This loose coupling between measurement and calculation and inside the calculation part supports reuse of calculation and measurement parts, which we already discussed in section 2.3.6.

ReD-IC2: Be flexible The most important challenge that our reference architecture must face is flexibility (challenge C1 from section 1.1.3). This is also emphasized in the top level requirement ReT-I7. Additionally, section 2.2.3 introduced our idea for metric variability. The aim of this is to add flexibility and reduce the number of metrics in a metric portfolio. The calculation part of the reference architecture, however, needs to reflect these concepts. As discussed before, the implications of our measurement data flow especially the implications from the satisfiability and compatibility relations defined in the formalism in sections 2.3.6 and 2.3.5 need to be reflected as well. In essence, the calculation part of the reference architecture needs to contain mechanisms to manage different variants of calculations and flexible connection between the calculation and measurement part.

ReD-IC3: Enable (near) real-time calculation Metric customer just interacts with the visualization of the measurement infrastructure. They can not distinguish which part is responsible for high latency between an update in a data provider and respective change of the visualization. Hence, performance and speed of the calculation is a concern when designing a calculation item and needs to be supported by specific mechanism in the calculation part of the reference architecture. As discussed with the measurement part above the requirement ReT-I2 requires mechanism to support “fast and up-to-date recognition and update of the metrics on a change in a data source”. The calculation part needs to include mechanisms to support this (if possible).

ReD-IC4: Respond to and report indication of data quality The items in the measurement part are able to indicate the data quality of specific measurements. The reference architecture for the calculation part needs to include mechanisms that allow calculation items to deal and react to the data quality. Additionally, they need to be able to report the data quality further up to the visualization part.

Requirements for Visualization

The visualization part contains all the tools that the metric customers interacts with. The items in the part visualize indicators from the calculation part and are specified by monitors (see terminology in section 2.1.1). The requirements listed below, however, just reflect the requirements to the visualization part of the reference architecture that are either intrinsic or originate from the interconnection with the other parts. As mentioned above, these requirements do not reflect the specific requirements for the tools that a metric customer interacts with!

ReD-IV1: Access data from different calculation sources The items in the visualization part need to be able to consume data (indicators) from a brought variety of sources in the calculation part. Hence, the visualization part of the technical reference architecture needs to provide a mechanism to uniformly access the different calculation items. Uniform access, however, does not require uniform data!

ReD-IV2: Be flexible Similar to the items in the calculation part, the items in the visualization part need to support different variants i.e. slightly tailored versions of the same visualization according to a variability model. Also, the technical part of the reference architecture needs to support mechanisms to reflect small changes in the information needs of the metric customers. Therefore, it needs to contain mechanisms to change the data sources (the indicators) of a visualization as well as the exchange of a visualization for fixed indicators.

ReD-IV3: Support dashboards and analysis tools The reference architecture needs to include mechanisms that allow to build dashboards as well as specific analysis tools based on the data in the calculation part. A dashboard provides a quick static status overview, typically for monitoring purposes, over a brought variety of very detailed and specific information needs. Consequently, monitors from dashboards are included in different status reports of the company as well. On the contrary, an analysis tool is only bound to very rough information needs. The focus of analysis tools is to investigate a specific situation and find (and answer) more specific information needs. Therefore, they typically provide a number of controls that allow to dig into the data and analyze a specific situation from different perspectives.

Requirements for the Infrastructure

A polylythical system based on the microservice architecture style requires additional tooling for the operation and management of the infrastructure. Operation support was also one of our top level requirements (ReT-I8). The following list of requirements again provide only top level requirements to operation tools and integration of those. The later sections 6.1 and section 6.2 provide more fine grained requirements in their initial sections.

ReD-I1: Separation of concerns The microservice architecture style, described above, requires small coherent services. Hence, the technical reference architecture needs to reflect this with mechanisms that enable the management and operation of multiple calculation items. The logical reference architecture needs to enable an information need driven design and separation of the different items in the parts.

ReD-I2: Report status of all items All items in all parts need to be manageable. An operator needs to determine whether a given item works within its defined operational boundaries. Operators also need an overview over all items in the infrastructure and their status. Hence, the reference architecture needs to include a monitoring tool and specific integration of monitoring in all items. More detailed requirements for the monitoring system are addressed in sections 6.1.

ReD-I3: Report operation information During development the developers need feedback from the operation of the items in their local measurement infrastructure. The items in the production version of the measurement infrastructure need to provide technical information to the operators and business information to the metric experts and metric customers. The reference architecture needs to address this with a specific logging solution for the measurement infrastructure. Again, section 6.2 provides finer requirements for the logging system.

ReD-I4: Robustness and evolution support The technical reference architecture needs to include mechanism to ensure the robustness of the resulting measurement infrastructure. It needs to include something to reintegrate (specifically calculation) items that stopped due to a failure or fault. Additionally it needs to support the evolution of the measurement infrastructure in a sense that it provides mechanisms to integrate new (calculation) items. Integration does not only require to connect them with other items but also ensures that the items have all the data that they need to ensure correct calculation of current and past measurements.

ReD-I5: Tool for metric documentation and interpretation aids management The literature lists metric documentation as an important success factor for metric systems. Additionally, in the terminology in section 2.1.1 we defined the need for dedicated interpretation aids for each monitor and information needs that they address to increase the usability of the metric system by guiding the interpretation by metric customers. Therefore, the reference architecture needs to include specific systems that provide metric documentation and aid the management of

interpretation aids. It also needs to enable easy integration between these systems and the items in the different parts (specifically the visualization part) of the measurement infrastructure.

3.1.3. Reference Architecture Requirements Summary

The section above added more specific details to our top level requirements from section 1.2. We started by introducing our main design decision for a polylythical measurement infrastructure based on the microservice architecture style rather than a monolythical solution. The main reasons for this are flexibility (challenge C1) and evolution support (ReT-I7) as well as separation of concerns (ReT-I3) on a high level. We then continued to list specific requirements for the core parts of the measurement infrastructure: measurement, integration, calculation, visualization, and infrastructure. The infrastructure part is a result from our main design decision. The most important requirements are again flexibility, real-time data, indication of data quality, and robustness. After listing these requirements, the following section will describe our API specification language that we use to define service APIs in the technical reference architecture.

3.2. The API Specification Language

For several services and their application programming interfaces (APIs) in the technical reference architecture we need to be able to specify the interfaces of these APIs on a formal basis. We need to be able to specify whether a given method or parameter is mandatory or optional. We also need to be able to parameterize a type with the interface for the definition of the indicator access APIs in section 5.3.2.

Several approaches exist to define interfaces using interface description languages (IDLs). For example the very formal approach from Arbab et al. [ABB00]. They use their formalism for component based systems as a basis. The idea behind their interface description language is to "introduce a formal assertion language for specifying the observable behavior of a component [...] via an interface". However, for our application this language and the underlying formalism is too complex to use. The web service description language (WSDL) is also used to describe interfaces and web APIs [CMRW07]. Unfortunately, the WSDL is specific and does not allow to distinguish between mandatory and optional methods and parameters. The common object request broker architecture (CORBA) includes the interface description language of the object management group (OMG IDL) [Vin97]. This IDL is used to define the types of operations on remote objects, similar to WSDL. Hence, like WSDL, it does not contain a mechanisms for genericity or conditionals for methods and parameters. Popular description languages for REST services like REST IDL also lack these options [MSW].

We, therefore, define a specific API specification language to define the requirements for specific API methods in our reference architecture. We designed the language to be easily readable and very narrowly focused towards our requirements. The following Backus-Naur form (BNF) defines the language:

⟨*interface*⟩ ::= '`interface`' ⟨*name*⟩ [⟨*typeParameter*⟩]
⟨*methodList*⟩

⟨*typeParameter*⟩ ::= '`generic type parameter`' ⟨*name*⟩

⟨*methodList*⟩ ::= ⟨*method*⟩*

⟨*method*⟩ ::= ⟨*conditional*⟩ '`method`' ⟨*name*⟩
⟨*parameterList*⟩
'`returns`' ⟨*type*⟩

⟨*parameterList*⟩ ::= ⟨*parameter*⟩*

⟨*parameter*⟩ ::= ⟨*conditional*⟩ '`parameter`' ⟨*name*⟩ '`:`' ⟨*type*⟩

⟨*conditional*⟩ ::= '`mandatory`' | '`optional`'

⟨*type*⟩ ::= ⟨*name*⟩ | '`List of`' ⟨*type*⟩

⟨*name*⟩ ::= ? any character ?

An API specification starts with the keyword `interface` followed by the name of the interface specification. After that we can specify an optional type parameter. This could be extended to a list of type parameters but we never needed more than one and we wanted to keep the language as specific as possible. If used the name for the type can be specified after the keyword `generic type parameter`. When the interface specification is instantiated this type parameter needs to be specified and every occurrence of the name needs to be replace by the specified type. After this we can specify the methods of the interface. The definition of a method starts with a conditional. This conditional indicates whether the method needs to be present in the API (`mandatory`) or if it does not need to be present (`optional`). The name of the method is specified after the keyword `method` followed by a list of parameters for the method and the return type after the keyword `returns`. The `void` keyword can be used as return value to specify procedures. A parameter, again, starts with a conditional followed by the keyword `parameter` and the name of the parameter. The type of the parameter is specified after a colon. Types are simply specified by their name. Lists are natively supported in the language with the keyword `List of` before the type name because they occur regularly.

The following example specifies a comparator API. The comparator requires a type parameter because we like to define the API regardless of the type of the elements that are compared. The comparator API requires a method `areEqual` to check if two elements are equal and a method `isGreater` to check if an element is greater than another one. Additionally the method `isSmaller` can be defined to check if an element is smaller than another one. This method is optional because it can be calculated from the result of the other methods (if not equal and not greater than it is smaller).

Source Code 3.1 API specification for a comparator API

```
interface Comparator
  generic type parameter DataType

  mandatory method areEqual
    mandatory parameter element1 : DataType
    mandatory parameter element2 : DataType
    returns Boolean

  mandatory method isGreater
    mandatory parameter element1 : DataType
    mandatory parameter element2 : DataType
    returns Boolean

  optional method isSmaller
    mandatory parameter element1 : DataType
    mandatory parameter element2 : DataType
    returns Boolean
```

The following Java-based JAX-RS service is a valid REST service implementation of the API specification from above.

Source Code 3.2 Java-based comparator REST service

```
@Path{"/compare.json"}
@Produces(MediaType.APPLICATION_JSON)
public class Comparator {
  @GET
  @path{"/areEqual/{element1}/{element2}"}
  public Response areEqual (
    @PathParam("element1") String element1,
    @PathParam("element2") String element2
  ) {
    boolean result = element1.equals(element2);
    return Response.ok(result).build();
  }

  @GET
  @path{"/isGreater/{element1}/{element2}"}
  public Response isGreater (
    @PathParam("element1") String element1,
    @PathParam("element2") String element2
  ) {
    boolean result = element1.length() > element2.length()
    return Response.ok(result).build();
  }
}
```

In this instantiation the type parameter is instantiated to `String`. Also note that the method `isSmaller` is not provided by the API. The two REST endpoints implement the mandatory methods. The `isGreater` method is implemented using the lengths of the strings. Also note that the language does provide a way to specify how the parameters are passed to the method and that the HTTP method type can not be defined[1]. The language also does not allow to specify the service technology (REST, SOAP, RMI, ...). These aspects are chosen when instantiating the API from our reference based on the requirements of the specific scenario.

The example shows the simplicity and readability of the API specification language compared to Java-based implementation of the Service. We use the language in our technical reference architecture when we need to specify specific APIs on the services.

[1]The example uses `GET` over `POST` because the methods are idempotent. In this specific case they are even nullipotent. Idempotent methods produce the same output whenever they are called. A nullipotent method is an idempotent method that does not change anything on the system.

3.3. Integration Architecture Alternatives

The integration of heterogeneous data sources is the main requirement of an enterprise measurement infrastructure (ReT-I1). Throughout the last decade several approaches have been proposed to deal with the emerging integration problems faced by developers and architects when dealing with heterogeneous systems and software landscapes. This research topic is known as: Enterprise Application Integration (EAI). In this section we present and analyze four existing types of enterprise application integration approaches and distributed measurement infrastructures based on our top level requirements from section 1.2.

File-based Integration

The simplest integration approach uses files to exchange data between applications. One application exports the data needed by another as a dedicated file. This file is imported by the other application and processed. This form of integration has certain disadvantages. Most importantly there is no real communications between the applications. Additionally, the export and import of data has to be synchronized. This directly violated the requirements ReT-I1, ReT-I7, ReT-I8, and most importantly ReT-I2.

Common Database

This integration approach uses a common database for all integrated systems. This provides a low latency to recognize and process relevant events. The main drawback is the necessity of an additional database management system. Successful implementations of this integration type are data warehouse systems. They provide an integrated database organized in a star schema [CD97], which includes multidimensional aggregated data cubes.

Integration via a common database or a data warehouse is the most common used integration approach chosen by recent measurement systems like Rational Insight. Sonarqube (`http://www.sonarqube.org/`) is another popular examples for a measurement tool that use a common database to integrate measurement data. However, as these systems are based on centralized data bases they directly violate requirement ReT-I3. Additionally, they are not able to guarantee requirement ReT-I2 and ReT-I7.

Service Oriented Architectures

Service-Oriented-Architecture (SOA) is also commonly used as the basis for integration solutions. The central idea of SOAs are systems following the service-paradigm [PTDL08, GDPG03]. They provide a stable self-describing interface for accessing internal data and functionalities. Web services are a prominent example for service-based software systems, which uses XML or JSON over HTTP for their communications. A popular example

for a SOA based measurement system is Hackystat (`https://code.google.com/p/hackystat/`).

Even thought SOA based integration is quiet common a pure SOA solution typically does not provide the required set of operation tools. Hence, this violates requirement ReT-I8. Additionally most of the integration solutions use SOA only for the communication between the visualization clients and a central data base server which violates requirement ReT-I6.

Enterprise Information Integration (EII) is a special case of SOA based enterprise application integration. The main goal of EII is to avoid a central database [HAB05]. EII adds a central query processor to an infrastructure of loosely coupled services. This central processor divides a query into sub queries to the services and aggregates their results. Even though this is an elegant solution to avoid a central database it violates requirement ReT-I4 and since the central processor needs to wait for all the sub queries to finish before returning an aggregated result it can take a while before the system answers which violates ReT-I2. Additionally, this still requires a central data schema in the query processor which violates requirement ReT-I6.

Agend-based Integration

Some modern integration approaches use agents to communicate between different applications [Woo01, KZR06]. Agents were first used in artificial intelligence systems [RNC+10]. An agent acts in a certain environment, uses sensors to get information about it, and can use this information for its decisions. Wille, Dumke et al. propose agent based measurement tools [DKW00, Dum12]. Even though agent based systems satisfy a large subset of the requirements they violate some of them. They typically do not provide central monitoring functionalities on their own. Furthermore, they are often hard to integrate into existing or new systems because of their very loose coupling. Hence, agent based systems violate requirements ReT-I7 and ReT-I8.

4

Logical Reference Architecture and Physical System View

Our reference architecture is divided into different parts: the technical reference architecture, the logical reference architecture, and the physical system view. Before defining the technical reference architecture in section 5 this section will introduce the logical reference architecture in section 4.1 and provide insides into the physical system view in section 4.2. The focus of the logical reference architecture is to guide the design of specific logical architectures for measurement infrastructures in a metric system. The physical system view will introduce the different systems in the context of an enterprise measurement infrastructure. It also presents a classification of the systems in the EMI core.

4.1. Logical Reference Architecture

Figure 4.1 provides a model for the logical architecture of a measurement infrastructure. The core of the logical view on an enterprise measurement infrastructure are different *Metric Components*. These metric components provide their required functions using a platform that contains specific frameworks, libraries, tools, and other reusable assets. We differentiate four types of metric components: *Visualization Frontend Components*, *Visualization Components*, *Calculation Components*, and *Measurement Components*. The later three originate from the core functionality of a measurement infrastructure (visualization, calculation, and measurement). Contrasting our definitions from before, the *Visualization Frontend Component* is added in the logical architecture as an abstraction for the applications that the metric customers interact with to look at the visualizations. In the architecture requirements we treated these as part of the "visualization". We separated them in the logical architecture because actual visualization components and concepts can be shared between different frontend components. As mentioned earlier, we differentiate between two types of frontend components: *Dashboard Tools* and *Analysis Tools* (requirement ReD-IV3). The relations between the metric components in the model reflect the data and control flow between them. Visualization frontend components utilize a number of visualizations. Visualizations get their data as indicators from calculations. They, as usual, receive their data from the measurements. We did not include the integration part in the logical architecture because it is a single technical part that is fix for all components and hence would only blow up this logical model without providing additional benefits.

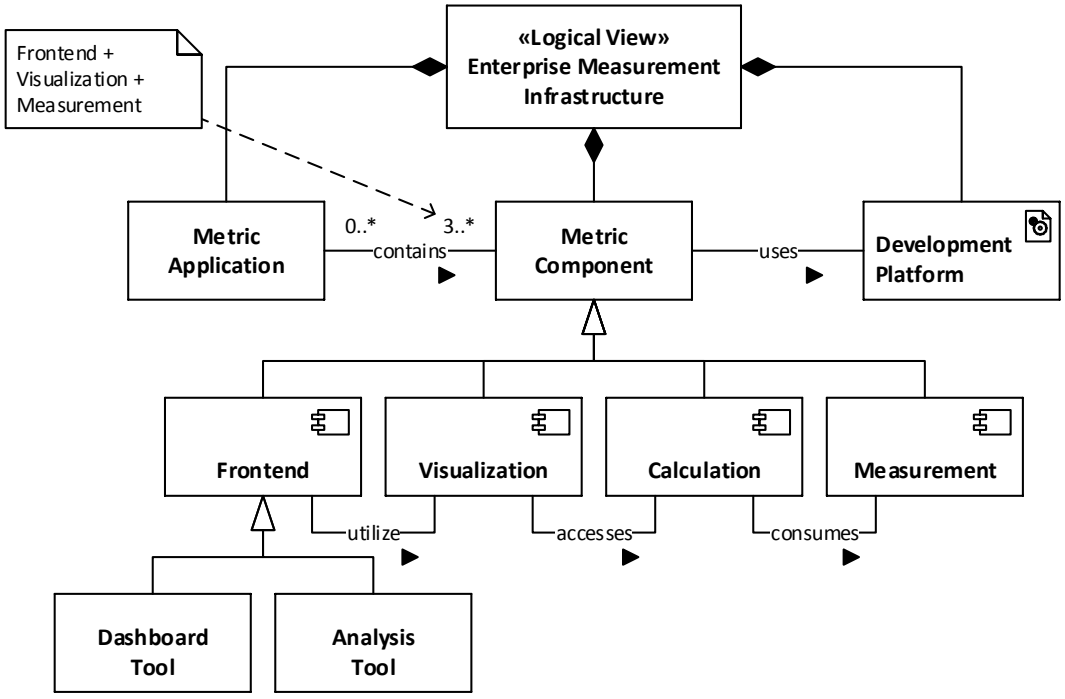


Figure 4.1.: Model for the logical reference architecture as UML class diagram

Examples for (logical) visualization components are bar chart, line chart, a risk matrix, or a table. Logical calculation components examples include components for counting, statistical analysis, or sorting. Generic examples for measurement components include generic data gateways and event triggers. However, in general measurement components are closely entangled with specific (types of) data providers and hence less generic.

During our field studies and the application of our reference architecture we realized that these logical architectures can become big and we need another item to structure the metric components. To do this we added Metric Applications to the model. They allow to decompose the logical architecture of a measurement infrastructure into smaller and easier manageable units. One metric application, however, always needs to include at least one frontend, one visualization, and one measurement component. A calculation component is not necessary required if the data from the measurements is sufficient for the visualization. The metric application associates its components with a (coherent) set of information needs of metric customers. This abstraction helps to manage component and information need traces which provide information to ensure separation of concerns (requirement ReD-I1). Therefore, metric applications are good guides and abstractions for a development increment in our process model.

Similar concepts are sometimes used to decompose large architectures into smaller parts. These parts are then called *architecture slice*. A slice, however, is a container that exclusively refer to their components. A metric application, on the other side, just references a number of metric components to provide a higher level of abstraction to

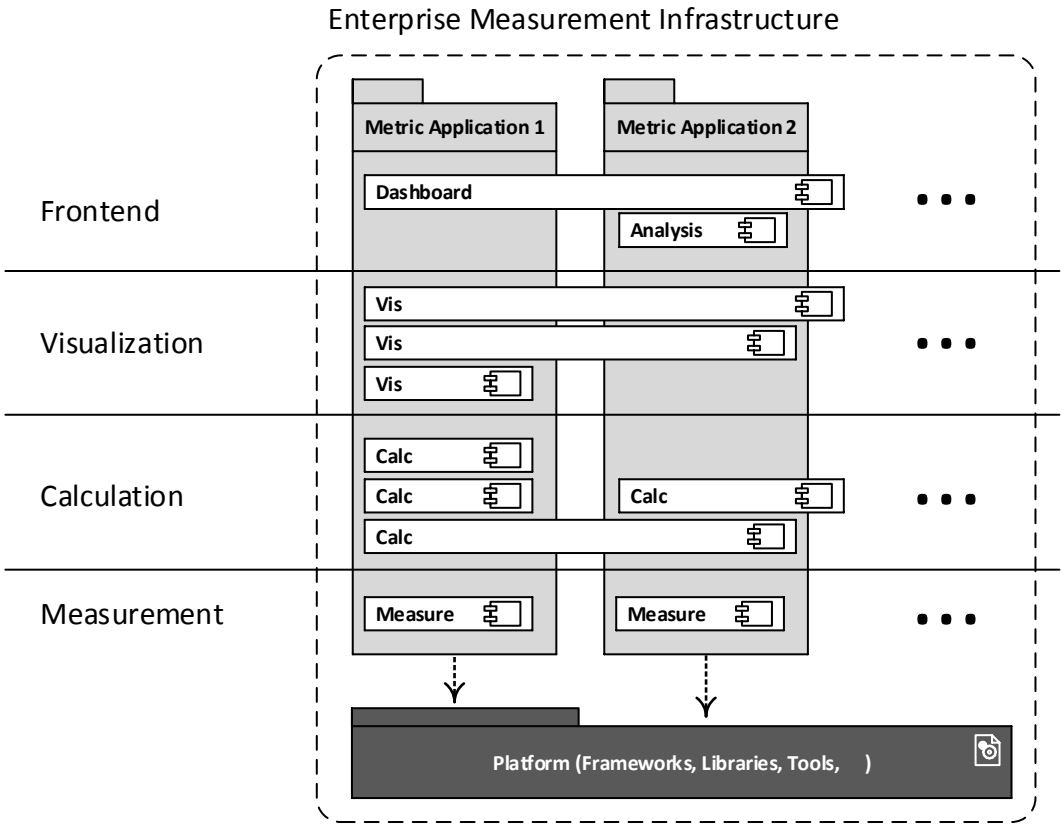


Figure 4.2.: Example for the logical architecture and logical decomposition of an enterprise measurement infrastructure

these components and this reference is not exclusive. Hence, the concept is more closely related to the composition of products in software product line engineering [Rom05]. One of the drivers of software product line engineering is reuse. In section 2.2.3 we already emphasized the importance of (metric) reuse. These ideas are also reflected here by the fact that metric components are not exclusively associated to a metric application but shared among them. Hence, when required to build a new metric application to answer a new set of information needs the architects can simply select fitting existing metric components and only add the minimal offset of new metric components required. This improves the quality of the application and saves development and operation resources. This design also adds a lot of flexibility to the composition of the metric system which supports a lot of our initial requirements (requirements ReD-IV2, ReD-IC2, ReD-I4, and ReT-I7) and tackles the most important challenge C1. Section 10.3 and section 11 in part III provide additional details on increment planing and the design part of the process model.

Figure 4.2 provides an example for a logical architecture and the logical decomposition of an enterprise measurement infrastructure. The component types from the meta model

result in corresponding layers with their respective components. The figure shows two metric applications that are both based on the common development platform in the bottom. Starting from the bottom both metric applications contain a single specific measurement component. Contrasting this, the bottom calculation component is shared between the two metric applications. Additionally, metric application 1 contains two specific calculation components. Metric application 2 contains one other calculation component that is shared with other metric applications. The two metric applications also share the top two visualizations. Again, metric application 1 requires one additional specific visualization. Both metric applications also share a dashboard tool as a frontend component. Metric application 2, however, also contains an additional analysis tool.

4.2. Physical System View

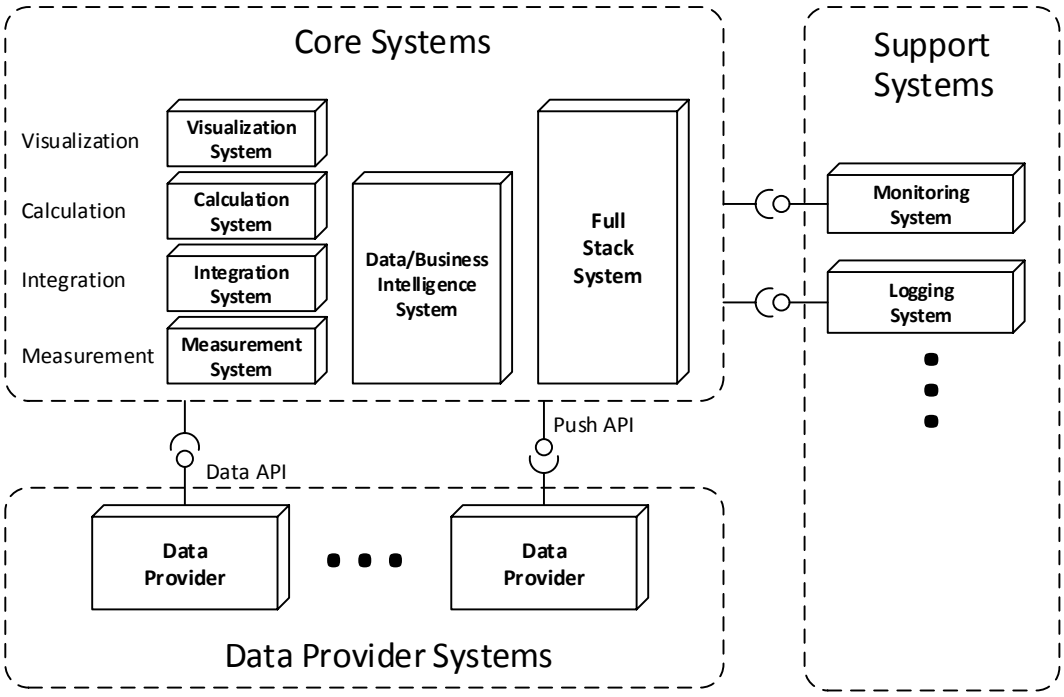


Figure 4.3.: System View

Figure 4.3 shows an overview and example of the physical system view on an enterprise measurement infrastructure. The data provider (systems) in the bottom provide the data required for the calculation and visualization in the measurement infrastructure. The right support layer houses all the support systems for the operation and development of the measurement infrastructure. The two most important ones are the monitoring system and the logging system. The measurement infrastructure reflects the different parts of the requirements: Visualization, Calculation, Integration, and Measurement. Contrasting the logical reference architecture, we did not differentiate the visualization layer further because the (logical) visualizations and frontend are typically provided as one physical system. Throughout this section we use the term *system* as a synonym for *physical system* to increase readability. The following sections provide additional details to the different systems.

4.2.1. Data Provider Systems

As the name suggests the data provider systems provide the base data to the measurement infrastructure. Data provider systems assist the business of the company and hence are used by different kinds of users. These users are typically not the metric customers that use the measurement infrastructure. Hence, it is important to ensure that the

measurements do not hinder the other users from using the data provider systems as intended (requirements ReD-IM3, ReD-I4, and ReT-I4). Consequently, the measurement should be as *invisible* as possible and provide the data automatically [Joh01].

In the technical reference architecture we differentiate different data adaption patterns to adapt the data from the data provider systems (section 5.4). However, we only need to differentiate between two different types of data access in the physical system view. The measurement infrastructure can either access the data from the data provider or the data provider provides the data to the measurement infrastructure on its own. The data access from the measurement infrastructure requires a data access API on the data provider. The measurement system (part) in the measurement infrastructure then uses this API to get the required data. If the data provider provides the data directly to the measurement infrastructure then this requires an API on the measurement system (part) of the measurement infrastructure. The data is then transfered (pushed) from the data provider using this API.

4.2.2. Support Systems

Operation support was one of our top level requirements (ReT-I8) and something that was and mostly is missing with other solutions. Therefore, we specifically included operation (and development) support systems in our reference architecture. Many of these systems also require a centralized element to provide their services. These then manifest in dedicated physical systems in this view. The two requirements ReD-I2 and ReD-I3 already imply two core support systems: The monitoring system and the logging system.

Monitoring System

The goal of the monitoring system is to provide status information about the other systems in the measurement infrastructure (see requirement ReD-I2). The most important status are:

Operation State The operation state of a service provides a high level aggregated view on the performance indicators of the service. The set possible operation states needs to be individually defined per system. However, all systems inherit at least an *online*, an *offline*, and an *unknown* state to indicate their core operation status. We investigate other important states (maintenance, data provider sync) further down in section 7.1.

Performance Data The systems in the measurement infrastructure need to operate within their defined operation parameters. Therefore, the operator requires the indication of performance data to check this against the operation parameters. Section 6.1 provides additional information about the most important performance data for measurement infrastructures based on our reference architecture.

Core Errors and Failures Errors and failures that lead to an operation state change should be directly indicated in the monitoring system to guide the immediate

actions of the operators. This may require to couple the monitoring and logging systems or to provide an additional communication mechanism for these details between the systems in the measurement infrastructure and the monitoring system.

Additional requirements and details on the design of the monitoring system are provided in section 6.1.

Logging System

The logging system is responsible for providing logging information from the services in the measurement infrastructure to a centralized service with uniform access. This drastically eases the development and operation of the different systems in the measurement infrastructure. The information provided to the logging system needs to be explicitly defined which is addressed in our process model in section 11. The two most important informations provided by the logging system are:

Technical Log The technical log provides technical information to the developer and the operator of a measurement infrastructure. Technical logging information supports debugging and root cause analysis. This can become difficult in distributed systems with only loosely coupled services.

Business Log The metric customers also require specific insides into the details of the data processing of their data. They may change data in a data provider and like to see corresponding visualization changes in a dashboard. If the data is mis-formated or contains inconsistent data, however, the calculation or measurement parts will reject the data. The metric customer then needs to be able to investigate the details of the rejection to correct the errors in the data in the data provider.

Additional requirements and details on the design of the logging system are provided in section 6.2.

4.2.3. Core Systems

We refer to the systems in the measurement infrastructure that perform metric related tasks (visualization, calculation, integration, and measurement) as *core systems*. The core systems can be classified according to their tasks as depicted in figure 4.3. We differentiate between the following classes of systems:

Measurement Systems provide abstraction and access to data from data providers. Examples for measurement systems are LOC counter or larger measurement systems like the Sonar Runner from Sonar Qube or ETL-engines like Kettle.

Calculation Systems calculate new measurements based on provided measurements. A very common example for calculation systems are Excel spreadsheets. Another example for a calculation system is the weka data mining toolkit and the OLAP system Mondrian.

Integration Systems integrate different data formats and sources between calculation and measurement systems. Popular examples for integration systems are Enterprise Services Busses like Apache ServiceMix.

Visualization Systems provide visualizations of calculated values from other systems. Again popular examples for visualization systems are spreadsheet and presentation tools, like Microsoft Excel and Power Point. However, there exist a large variety of dedicated visualization systems for measurements like GGobi and Data-Driven-Documents: D3.

Data/Business Intelligence Systems abstract and access data and perform measures on these data. Data Warehouses systems like apache hive are a typical examples for these kind of systems.

Full Stack Systems implement all tasks. Popular examples are Pentaho and Cognos/Rational Insight.

The reference architecture is applicable for all of the above types of systems. It also assists the integration between the different parts and helps to include the support systems. Heterogeneous infrastructures with different types and varieties of core systems are also supported. Obviously, we assume our foundations (see section 2) as a basis for the reference architecture. Hence, it is more difficult to include a full stack BI-system and gain all the benefits from our design then to include an enterprise service bus for the integration part which already fits well to our design.

The last two sections defined our logical reference architecture and physical system view on measurement infrastructures the next section presents the core of this part: our technical reference architecture.

5 Technical Reference Architecture

This part started with the definition of the detailed requirements and core design decisions for our reference architecture for enterprise measurement infrastructures in section 3.1. Section 4 then defined our logical reference architecture as well as an overview over the physical system view and a classification of the core systems of an enterprise measurement infrastructure. This section defines the technical reference architecture on top of these important foundations.

The section starts with a broad overview over the different parts in the technical reference architecture in section 5.1. We define the layers and provide a brief introduction to the core components. After that we start with the first integration layer of the technical reference architecture between calculation and measurements in section 5.2 because this glues these two core parts together. After that we provide insides into the reference architecture for the second integration layer between the calculation and visualization layer in section 5.3. We continue with the reference architecture for the components in the measurement layer in section 5.4. With this we also define different data adaption patterns for a brought variety of measurement scenarios. Section 5.5 defines the reference architecture for the components in the calculation layer. The core layers are finished in section 5.6, which provides the reference architecture for the components in the visualization layer of the enterprise measurement infrastructure. After we closed all these core layers, section 5.7 will define a reference architecture for the integration of the operation system which is used by the monitoring system (section 6.1), the logging system (section 6.2), and the lookup service (section 6.3). We close the chapter with a summary in section 5.8.

5.1. Overview

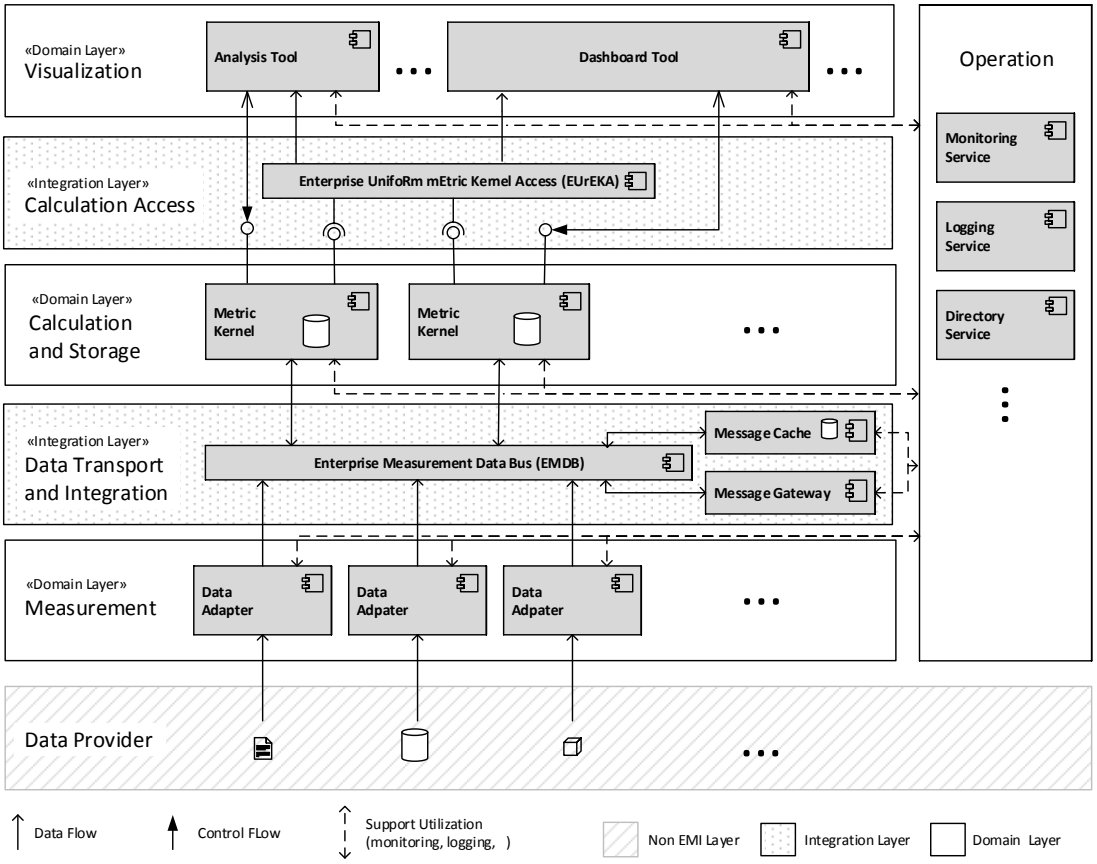


Figure 5.1.: Technical layers and services in an enterprise measurement infrastructure

Figure 5.1 provides an overview over the layers and components in our technical reference architecture. This figure mirrors the overview of the physical system view in figure 4.3. The bottom layer (data provider) and the right layer (support) are identical. The core, however, is enlarged and filled with additional components. We differentiate between two different types of layers: *Integration Layers* and *Domain Layers*. The integration layers, as the name suggests, glue their adjacent layers together. Therefore, these layers mostly contain architectures for concrete services and solutions to solve typical concrete (integration and maintenance) problems. The domain layers, on the other side, contain the actual functional components of the EMI. Therefore, we often just provide patterns and best practices as reference architectures for the components in these layers. They guide the design of the actual components when instantiating the reference architecture.

Before we go into further detail we first provide a quick overview over the components and their interactions in the following description of the layers from bottom to top:

Measurement The measurement layer houses the data adapters which implement the measurements from the metric portfolio. They specifically ensure requirement ReD-IM3 to isolate the data providers from the measurement infrastructure. The goal of the data adapters is to provide (access) data from heterogeneous data sources into the measurement infrastructure. This provisioning requires accessing data from a broad variety of data providers (requirement ReD-IM1) as well as a data transformation (data integration) from the data format of the data provider into a data format suitable for the integration in the measurement infrastructure (requirement ReD-II1). The design of the data adapters need to keep latency (time between data change and measurement) as low as possible to ensure requirement ReD-IM2. Section 5.4 provides a micro reference architecture for data adapters as well as several data adaption patterns.

Data Transport and Integration This layer contains the *Enterprise Measurement Data Bus (EMDB)*. As the name of the layer suggests the EMDB transport measurement messages as data from data adapters to and between metric kernel (requirement ReD-II2 and ReD-II4). The EMDB also needs to support the integration of heterogeneous data formats from the different measurement producers (requirement ReD-II1). Steffens already discussed the EMDB design based on the criteria for integration from Hohpe and Wolf in his thesis [Ste13] (requirement ReD-II3). Further details on the EMDB and the data integration are presented in section 5.2.

Calculation and Storage The metric kernels contained in this layer are the core of the measurement infrastructure. They implement the calculations, consume the data from the EMDB, persist important data, and provide indicator to the visualization layer. Hence, this fulfills requirement ReD-IC1 (separation between measurement and calculation).

The metric kernels, like all other components, are realized as microservices. This is particular important for the metric kernels as this enables free choice of the persistence technology (database), which drastically increases the flexibility in the design of the metric kernels (requirement ReD-IC2). Furthermore, this can help to optimize the performance of the metric kernel to allow very fast, near real time, calculation of the metrics (requirement ReD-IC3). The calculation of metrics in the metric kernel is typically performed on request from a visualization. The calculations are accessed through indicator access APIs provided by the metric kernel. These APIs also provide data quality indicators (requirement ReD-IC4). Section 5.5 provides a dedicated micro reference architecture for metric kernels.

Calculation Access This layer integrates the visualization and calculation layer by orchestrating the loosely coupled indicator services from the metric kernels. By implementing the requirements of our *Enterprise Uniform Metric Kernel Access (EUrEKA)* concept the indicators provided by the metric kernels can be easily accessed by the visualizations (satisfying requirement ReD-IV1). The main idea of the design is a specification for indicator access APIs as well as a description model for them on the metric kernels together with a registry service for their orchestration. This design provides more flexibility and coherence than a fixed uniform data format, which is typically used for the communication between calculations and visualizations. Section 5.3 provides additional details on the required services and on the indicator API design for metric kernels.

Visualization The top layer houses the visualization frontends similar to typical 3-tier designs [Eck95b]. As discussed in the logical reference architecture (section 4.1) this contains the two different types of frontends: Analysis tools and dashboards tools (requirement ReD-IV3). It also houses the implementation of the logical visualizations components. The goal of the visualization frontends is to provide monitors of the calculated or measured values. The frontends, on the other side, provide management functionality for and interconnection between the monitors. Section 5.6 provides a micro reference architecture for the frontends.

Operation Similar to the physical system view, the operation layer houses operation support services to the core measurement infrastructure. Besides the already introduced monitoring service and logging service it may include additional services like the *Lookup Service*. We provide a specific architectural style in section 5.7 for the integration between the EMI services and the operation services in this layer. The actual definition of the reference architecture for the operation services is provided in chapter 6 after this technical reference architecture for the core services.

The data provider in the bottom are not part of the measurement infrastructure (see physical system view). However, we included them in the overview because some data adaption patterns require do add a specific plugin to the data provider. The inclusion also increases the readability of the data flow. The technical reference architecture for the data adapter uses a fill-pattern to visually indicate the parts of the data provider that do not belong to the reference architecture.

5.1.1. Measurement Data Flow

The EMI data flow depicted in 5.2 is an extension of our conceptual data flow presented in figure 2.4 in section 2.1.2. This already extended the ISO 15939 measurement data flow by the means of circular data flow between derived metrics. The technical systems message cache and message gateway further extend the data flow. Similar to our conceptual data flow the data enters the measurement infrastructure in the bottom through measurement by the means of the data adapters. Hence, they provide base measures to the measurement infrastructure. Additionally, we added a message gateway in the bottom which allows

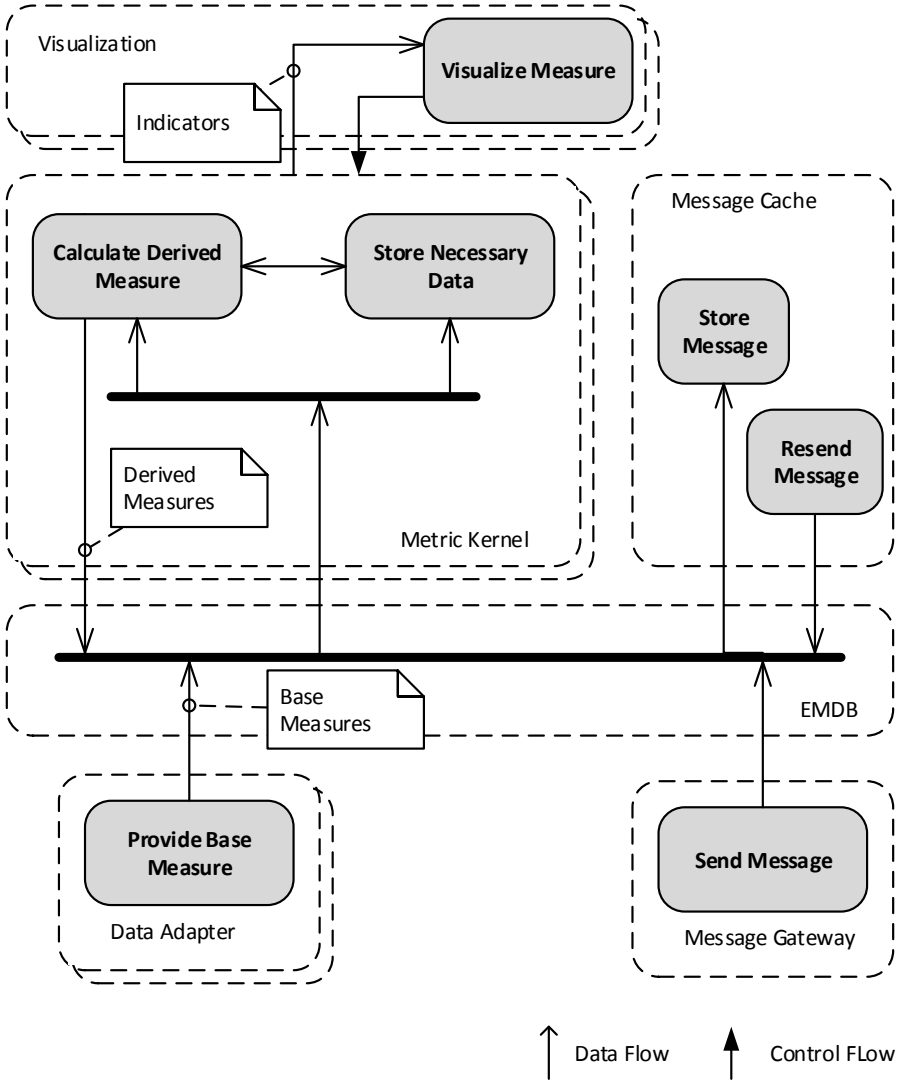


Figure 5.2.: Simplified data and control flow in an EMI

to send arbitrary measurement messages to the EMDB. This dramatically eases the integration test of metric kernels. On top of the EMDB the measurement messages are consumed by the metric kernels of the measurement infrastructure. They store some data which they require to calculate their metrics. Additionally, they produce new measurement messages which are resend to the EMDB. The metric kernels also provide indicators for the visualizations. This is again similar to our conceptual data flow from before. Additionally to this technical data flow, the measurement messages are also stored in a message cache. This is again useful for testing (measurement producers) and for operation tasks like setup of new or faulty metric kernels. Therefore, the cache is able to resend specific measurement messages to the EMDB.

Discussion on the Data Flow

Due to the design of the data flow metric kernels are not able to communicate with each other. They only receive data from the EMDB and can not exchange variability configurations. A request-reply based metric calculation would allow metric kernels to exchange variability configurations in order to calculate explicit derived metrics. However, the metric kernels would then need to formally define these in addition to the accepted types. This would complicate metric kernel reuse, metric kernel specification, and metric kernel design. Furthermore, it would increase the load on the EMDB. It would also contradict our push-based design in the integration and transport layer. Therefore, we decided to not allow request-reply-based metric calculation in the reference architecture. In a pragmatic, specific scenario, however, a metric kernel that (desperately) requires specific calculation results from another metric kernel could simply access the metric value access API of the other metric kernel in order to get the calculation result according to a specific variability configuration. This would, however, violate the communication principles from the reference architecture. Hence, a better solution for this would be to alter the requirements for the dependent metric kernel to include the derived metrics required by the original metric kernel and include this in the next development iteration.

5.1.2. Concept to Implementation Mapping

Figure 5.3 provides the mapping between the metric concepts presented in section 2.1.1 and their corresponding implementations in an enterprise measurement infrastructure. As defined earlier, metrics identified by a unique metric reference consists of a measurement approach. These approaches are specialized by measurement definition and calculation definition. The later consists of a measurement function and refers to other metrics which provide the inputs to the function. Visualizations contain a number of interpretation aids which describe how to read and interpret the visualization. They are also connected to the information needs of the metric customer.

In the enterprise measurement infrastructure the metric customers derive answers to their information needs from monitors. These are defined by the visualizations and hence connected to the interpretation aids. Monitors utilize a renderer to visualize the data provided by metric kernels. These provide the data by the means of a EUrEKA conform data access API. Metric Kernels implement a number of calculation functions whereas the data adapters implement measurement functions. Metric kernels and data adapters both utilize message senders which implement measurement producers defined in section 2.3.7. Consequently, the measurements are implemented by measurement messages which can be received by metric kernels.

5.1.3. Discussion and Design Alternatives

In this sub section we like to discuss alternatives to the layers and general layout of the technical reference architecture presented above. We mainly distinguish between two mayor alternatives: ad-hoc (unstructured) loosely coupled distributed systems and single

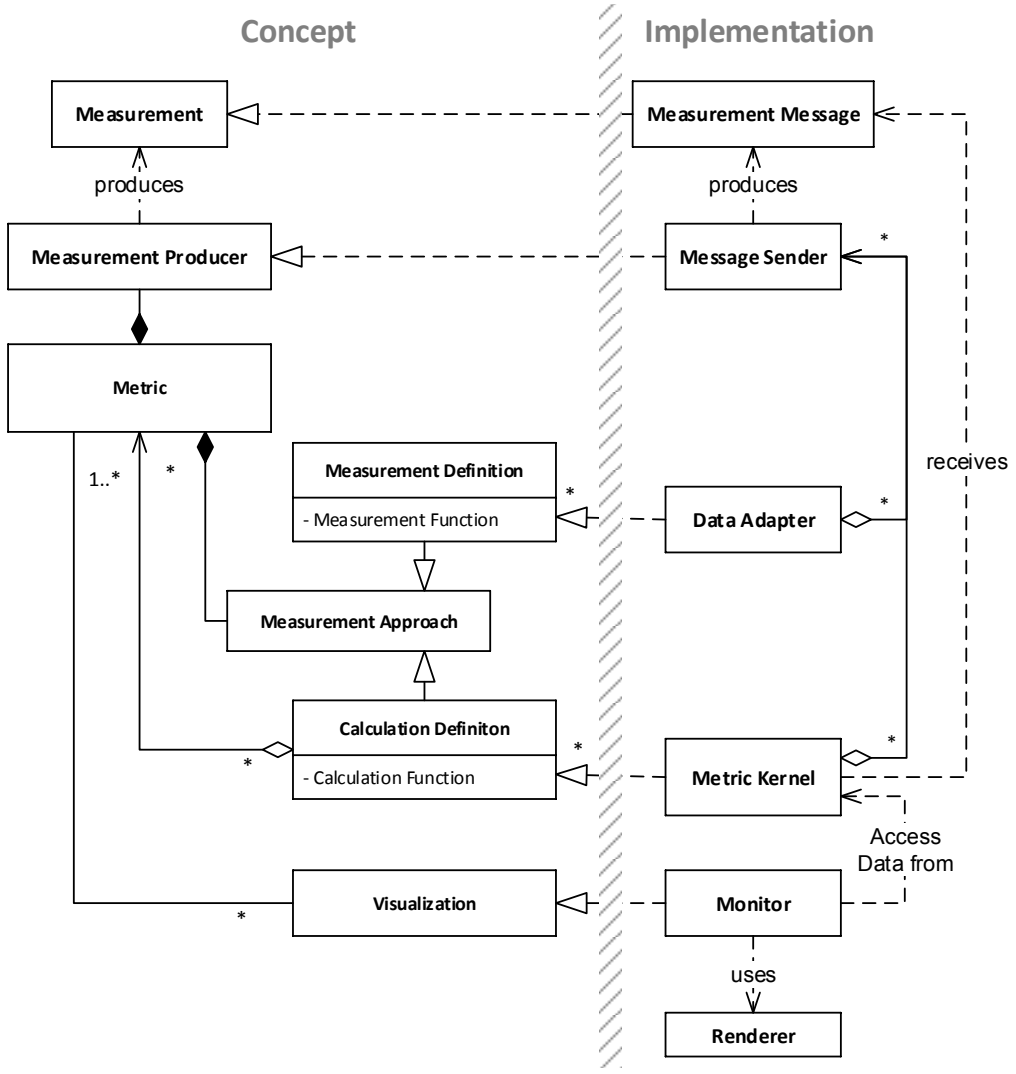


Figure 5.3.: Metric concepts and their corresponding implementations in an enterprise measurement infrastructure

database systems (e.g. data warehouses). The following sub sections provide additional details to the alternatives and additional discussions on their strengths and weaknesses.

Loosely Coupled Distributed Systems

We choose this as an alternative because it is a popular approach in our related work (see section 1.4). Also it is a common approach in the field of enterprise application integration which is supported by a lot of experiences and several well proven pattern [HW03a, Kai02, Hor14, Cha04, Fow02, Tho05]. It is also the approach chosen by the popular measurement tool Hackystat that we already mentioned above. Furthermore,

the concepts of Kunz et al. and Heidrich et al. that we discussed in section 1.4 both propose a SOA-based measurement infrastructure [KSDW06, Kun09, HM08a, Hei08]. The idea in both approaches is to provide a set of measurement services that support the main measurement use-cases: measurement of base metrics, calculation of derived metrics, and visualization. All these services are (relatively) loosely coupled to provide flexibility. Besides the design and definition of the services and their alignment in the overall improvement model the work from Kunz et al. does not provide any additional structure like layers or pattern.

The work from Heidrich et al. proposes a layered architecture [HM08a]. The layers for the the high level architecture of their reference implementation *Specula* are: *Data Collection*, *Data Processing*, and *Data Exploration* (*Data Visualization*). This closely reflects the layers in our reference architecture. However, their design lacks the two, as we believe very important, integration layers. The services in the layers are very generic. The core of the data collection layer is a repository manager service that monolithically orchestrates and performs the measurement. The data processing layer contains "Core Services" that implement the derived metrics. The data exploration layer contains the different dashboards. All these services are designed monolithically using a single database as the integration tool. Therefore this approach also falls into the alternative that we discuss in the following sub section.

The idea of these designs, however, can be extended in order to further support flexibility with multiple databases in the services (following the microservice concept). Therefore, a set of loosely coupled (unstructured) SOA services is a potential alternative to our reference architecture. The services are extended and connected on-the-fly when the measurement system needs to be changed based on changes in the metric portfolio or new tooling requirements. Instead of SOA-oriented services the different services, especially the measurement part, could also be implemented using distributed loosely coupled agents. The idea is similar; only the integration between the services is different.

The most important benefit of this design would be its flexibility. The lack of structure especially the lack of fixed layers, boundaries, and integration concepts allows to solve all requirements with ad-hoc solutions. Therefore, the implementation of such a measurement infrastructure will be faster than an infrastructure based on our reference architecture (an EMI).

The most obvious weakness follows directly from this argument: The lack of structure can lead to a messy architectures which generate higher maintenance costs. The longer such a system is maintained the higher the refactoring and restructuring effort required to integrate new requirements will be. Therefore, in the long shot the implementation in such a measurement infrastructure will slow down significantly over an implementation in an EMI. Furthermore, our reference architecture already provides a good documentation of the concepts of the different layers and their services. This contrasts the increasing high effort to keep the (individual) documentation of a SOA up-to-date. Therefore, the documentation effort in an EMI is lower and new staff will faster be able to work productively. Additionally, our reuse oriented metric management approach is reflected in a lot of concepts in our reference architecture. Therefore, multiple EMIs are able to

easily exchange and share services with very little additional effort; contrasting reuse of services in a SOA. Reuse of services will lead to higher quality of the services and, yet again, lower maintenance costs.

Single Database Systems

Single common databases and data warehouses are the most common used basis of large industrial measurement infrastructures. A single measurement database is also the recommended design by measurement best practices and listed as a measurement success factor [NV01, Pau06, HMO08]. However, we already discussed a lot of weaknesses with this design in section 1.2.1, section 1.4, and section 3.1.

The most important benefit of an architecture based on a single measurement database and more particularly using a data warehouse are their wide adoption in industry and academia. The architectural-basis for data warehouses and the database design is covered in education and is generally considered common knowledge. Therefore, the development of data warehouses is supported by a large variety of processes and supporting tools. The measurement mechanism into a warehouse using the extract-transform-load (ETL) mechanism is also known and heavily supported by tools. For example by the means of graphical notations to specify the ETL. Furthermore, data warehouses by the means of business information systems are successfully used in most companies. Most of the benefits, however, result from their focus on homogeneous financial data.

We already discussed a lot of weaknesses of measurement systems based on a single database (schema). Most importantly, they are very inflexible once they become large. The reason for this is that almost every new requirement needs to add or modify some part of the data schema. Schema transformation on large databases, however, is far from easy and generally requires a lot of effort for development, data migration, testing, and release.

Furthermore, a single database is obviously not robust against failure. If the one database is offline, for example for maintenance or because of a failure, then the complete measurement infrastructure will not be operational. Therefore, operators need to take additional precautions to guarantee robustness. For example mirroring the database and abstracting the extra database using a reverse proxy or load balancer. However, this also drastically increase maintenance effort (updates on multiple system, synchronization, testing, ...).

Managing and working with ETLs can get difficult as well. Our experience on ETLs in practice is that they will become large and complex. Especially metrics to monitor the software development process will get complicated. Like all large and complex software development artifacts: Large ETLs produce high maintenance effort and potential errors. Another issue with an ETL-based measurement mechanism is its timing. ELT is, almost always, triggered periodically. Therefore, the time difference between changing some data and seeing the result of the update in the measurement system (latency) is high.

Discussion on the MeDIC Reference Architecture

This subsection will discuss the benefits and weaknesses of EMIs based on our reference architecture. We derived these from the application of the reference architecture and numerous discussions on the topics with architects and users of the reference architecture.

The strongest weakness of our reference architecture is a high initial effort that needs to be invested before the first EMI is operational. A lot of the services from the reference architecture as well as a development platform needs to be build and established before development on the actual EMI services may start. If they are established, however, they all help to reduce the further development and maintenance effort significantly. Most studies see the relation at around 20% initial effort versus 80% maintenance effort over the complete life cycle of the software system [Sta03, BR00, Jon06, BDKZ93]. Therefore, we believe it is better to support effective and efficient maintenance over fast initial development. Furthermore, once the development platform is established and the services exist they can be shared and distributed. They can then be reused by other branches or even other companies to speed up the initial development of their EMIs.

EMIs based on our MeDIC reference architecture combine the two strengths from both approaches: They are very flexible and their development and application is supported by a wide variety of tools because we use well known and established techniques. Furthermore, our reference architecture proposes a large set of additional services that support the development, testing, and operation of an EMI. Furthermore, our reference architecture also helps to avoid most of the problems with the alternatives from above. The reference architecture provides enough structure and a good harness to avoid messy architectures and reduce maintenance costs. Furthermore, an EMI based on many microservices with independent databases and free choice of technology avoids the problems with single database measurement infrastructures. Contrasting graphically or DSL configured ETLs, our data adaption is programmed in data adapters. Therefore, developers, QA, and operators are able to choose among a large variety of processes and tools, which they are already familiar with, to identify and deal with potential maintenance problems.

This concludes our discussion and initial overview over the technical reference architecture. The next section will introduce the backbone of our reference architecture: the data integration and transport layer with its key part the enterprise measurement data bus (EMDB).

5.2. Data Transport and Integration

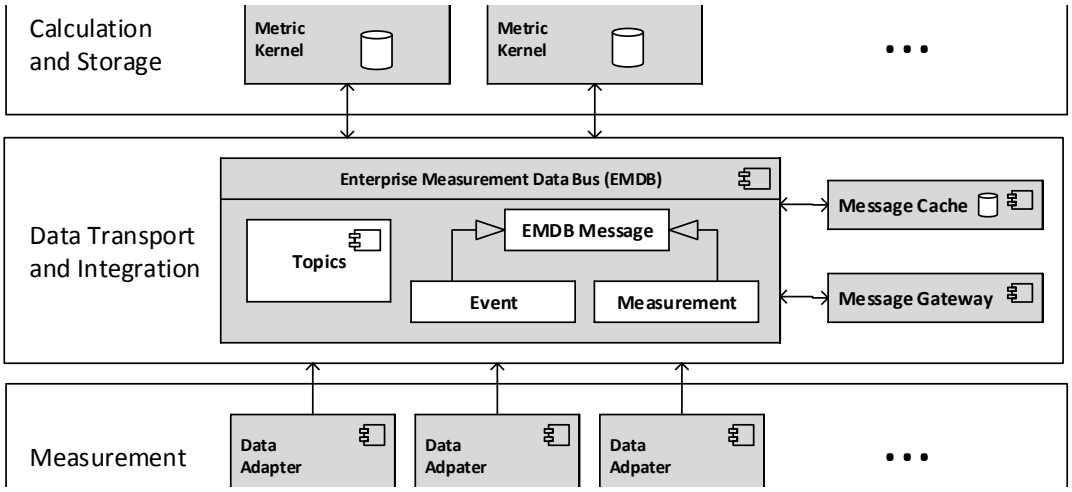


Figure 5.4.: Zoom into the data transport and integration layer of the MeDIC reference architecture

The last sections gave an initial overview over our technical reference architecture for enterprise measurement infrastructures. This section provides additional insides into one of the two integration layers of an EMI: the transport and integration layer. Figure 5.4 shows an overview over the components in the layer. The main concept is the enterprise measurement data bus (EMDB) in the middle. It contains several topics for the different integration and communication tasks between the measurement and the calculation layer. These topics exchange EMDB Messages which get specialized for different types of messages. Furthermore, the layer contains two support services on the right hand side of the EMDB: The *Message Cache* and the *Message Gateway*.

In this section we first describe the enterprise measurement data bus (EMDB) in the following subsection 5.2.1. After that we provide a detailed definition of measurement messages and their implementation in subsection 5.2.2. The reference architecture should support our engineering approach and accordingly we also need to support tasks like testing and development. Therefore, we added additional services to the integration and transport layer to support these tasks. Subsection 5.2.4 provides additional details on these services and how they support the tasks. The last subsection 5.2.5 introduces additional topics that are not directly involved in the measurement process but provide a communication infrastructure for additional services and functionality. We close the section with a short summary in subsection 5.2.6.

5.2.1. Enterprise Measurement Data Bus (EMDB)

The primary goal of the enterprise measurement data bus is to transport different types of measurement data from the measurement producers to the measurement consumers (requirement ReD-II2). Hence, it needs to accept measurement messages from data adapter and metric kernels and transport it to the metric kernels that consume the data. The bus, therefore, provides the integration between data providers, via data adapters, and metric kernels.

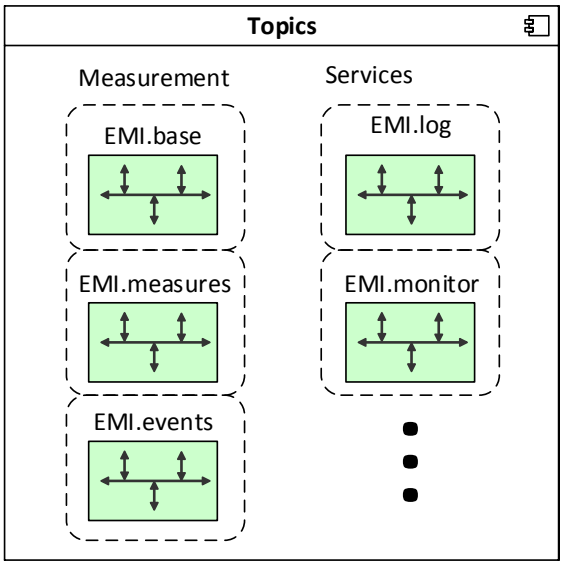


Figure 5.5.: Publish/subscribe topics inside the EMDB

Figure 5.5 shows the internal publish/subscribe topics (see Hohpe and Wolf for details [HW03a]) which implement the EMDB. This design follows the principle of separation-of-concerns with specific topics for the different types of data.

On the top level we differentiate between measurement topics and service topics. The measurement topics transport the measurement related data of the EMI whereas the service topics transport other data for services that are not directly related to measurements (like logging and monitoring). The following list provides a quick overview over the standard measurement topics:

EMI.base Transports the raw, and typically unprocessed, atomic data from the data adapter to the metric kernels. However, this can also transport processed data from a metric kernel to other metric kernels if this data is not a pure measurement or an event.

EMI.measures This topic transport the measurement data. Measurement data always contains one measurement value as a number! We recommend a specific topic for these types of data because they are very common in a measurement infrastructure and can be consumed by a lot of standard metric kernels.

EMI.events This topic transport event-based data. *Event data* indicates an action on data in a data provider, whereas *base data* typically is the result of such an action. Hence, contrasting base data which can represent *anything*, an event has a defined semantic. Therefore, we recommend using a specific topic for this type of measurement data. Also, like measurement data, event data can be consumed by standard metric kernels to calculate typical metrics like statistical event counting or statistical analysis of the event occurrences.

This design allows the metric services to just connect to the specific topics for the data that they require. Also, this isolates the different parts even further than just one topic (requirement ReD-II4). Obviously, these topics can be extended when instantiating the reference architecture. The following section provides details to the messages exchanged over these topics.

5.2.2. EMDB Messages

EMDB messages are the objects which are exchanged over the measurement topics on the EMDB. The service topics also require messages but due to their specific nature they are not bound to the requirements for measurement messages.

Following our formalism in section 2.3, which is extended by the reference architecture concepts in section 7.2, the messages on the topics of the EMDB are typed. However, as stated in requirement ReD-II1 the EMDB needs to deal with heterogeneous data types. Hence, the architect of the EMI needs to be able to model different types of data included in EMDB messages and integrate those with each other. We recommend to utilize generalization and specialization on these types to easily implement the concepts behind the compatibility and satisfiability relation from the formalism. This also helps to model and integrate complex data[1].

Figure 5.6 shows a possible specialization hierarchy for EMDB messages. The `Base Message` on the top is most general message type in this example. It contains fields for the data change timestamp and the entity of measurement. In this example it also contains a method to get the measurement identification (M_{Id} from the formalism). This base messages are exchanged over the `EMI.base` topic. Additionally, we recommend to define base messages for the two other measurement topics: measures and events. In this example the `Base Measurement Message` and the `Base Event Message`.

In this example we extended the Base Message with two specific base messages for data from a ticket management system (`Ticket Message`) and a version control system (`VCS Message`). These specific base messages add additional fields to the message for the data that they represent. In a real application these would then be extended by messages specific to the the data provider. For example a `JIRA Message` as a sub type of `Ticket Message` to represent tickets from a jira system. The following section provides further details into message sub typing and how it can be used to reuse metric kernels.

[1] See the diploma thesis of Steffens for additional details [Ste13]

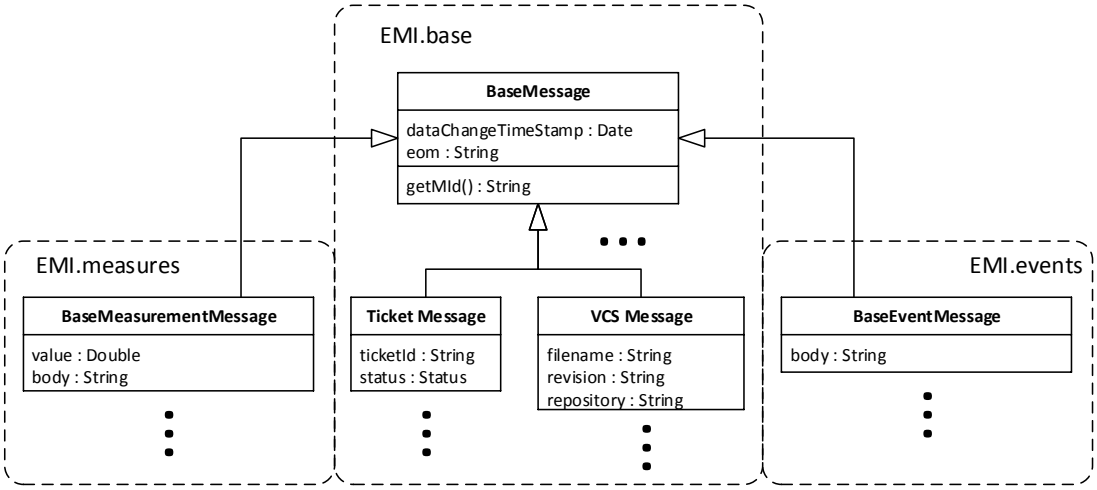


Figure 5.6.: Example for a EMDB message specialization hierarchy and relation to EMDB topics

5.2.3. Integration and Reuse

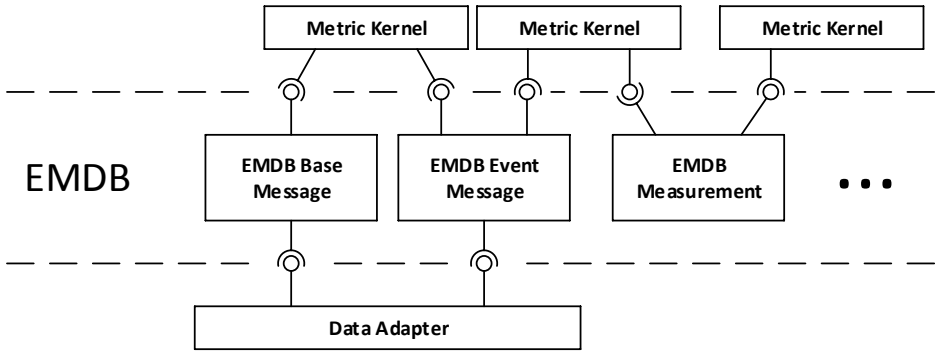


Figure 5.7.: Integration between Data Adapter and Metric Kernel via EMDB Messages

Figure 5.7 shows the conceptual integration of one data adapter and two metric kernels. The data adapter provides two measurement messages: a base message and an event message. Both are required for a calculation in the first metric kernel. The second metric kernel just consumes the event message and provides another measurement message, which is consumed by the third metric kernel. As visualized in the figure, the EMDB messages connect the provided data from the measurement producer to the required data from the measurement consumer and therefore integrate the two with each other.

In our formalism in section 2.3 we already mentioned different abstraction levels for metrics and potential reuse scenarios for derived metrics in section 2.3.6. We support this in the technical reference architecture by specialization hierarchies as described in the section above. Figure 5.8 shows an example for the reuse of a general metric kernel

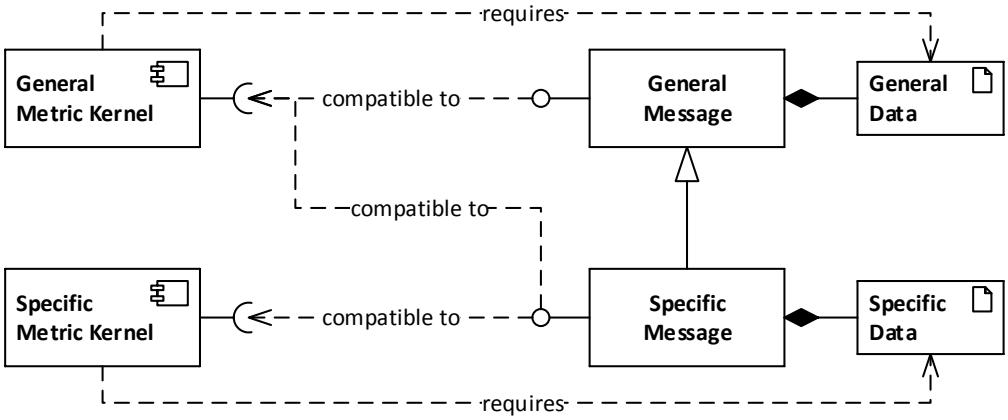


Figure 5.8.: Reusing a general metric kernel with specific EMDB messages

can be reused in a specific context. The calculation in the `General Metric Kernel` requires `General Data`. This general data is embedded in `General Messages`. The `General Message` is therefore compatible to the guard of the measurement consumer of the metric kernel. Additionally, the EMI in the example also requires a `Specific Metric Kernel` for other calculations. This metric kernel requires `Specific Data` which is included in `Specific Messages`. These specific messages, however, are specializations from the general messages. Therefore, they also include general data. Hence, specific messages are also compatible to the guard of the measurement consumer of the general metric kernel. Thus, the general metric kernel can be reused in this specific scenario and consumes specific messages.

5.2.4. Important EMDB Services

The following two services are also included in the data transport and integration layer. They drastically reduce (or even enable) maintenance and test effort in an EMI. Without these services the correct operation of the EMI is very hard to check. Consequently, without them the EMI will suffer from maintenance and quality problems. We first describe the reference architecture for a Message Gateway which provides an API to directly send EMDB messages over the topics without using a data adapter. This is particular useful for testing. After this we provide a reference architecture for the message cache. This cache, as the name suggests, caches all messages on the EMDB. Hence, it provides important internal details on the data flow of the EMI. Additionally, it provides mechanisms to resend particular messages or chunks of messages which is a mandatory feature for several maintenance tasks.

Message Gateway

When (integration- and system-) testing a metric kernel we require a mechanism to test the metric kernel in "isolation" without a data adapter or other metric kernel that provides the required data for the metric kernel under test. Thus, we require a mechanism in an EMI to send measurement messages directly to the EMDB without using a data adapter or changing data in a data provider. The Message Gateway Service provides this mechanism. An automated (or manual) test can then use the API of the Message Gateway to send test data to the metric kernel and check the calculation results using the EUrEKA API of the metric kernel (see section 5.3 and section 5.5). The Message Gateway can also be used when prototyping an EMI to simulate a data adapter or simulate the calculation of a metric kernel.

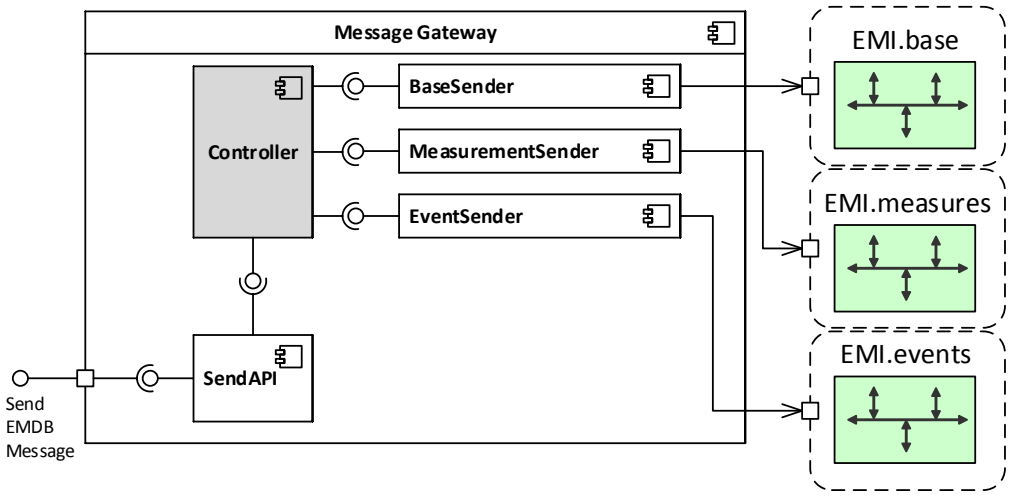


Figure 5.9.: Message Gateway internal component view

Figure 5.9 provides an overview over the internal components required to build a Message Gateway. The Message Gateway provides a send EMDB Message API to the outside world. The API is typically implemented as a SOAP or REST web service. Inside the Message Gateway it is realized by the `SendAPI` component. This component implements and exposes the web service. It then delegates the sending to one of the dedicated message senders. Typically each topic requires a unique sender. However, the message senders are most likely provided by the platform developed underneath the specific EMI because they are require by many services. The send API only needs to provide one send method but can optionally provide more specific methods as described by the following listing using our API description language:

Source Code 5.1 Send API of the Message Gateway

```
interface SendAPI

  mandatory method send
    mandatory busIdentifier : String
    mandatory message : String
    returns Void

  optional method sendMeasurement
    mandatory metricId : String
    mandatory value : Double
    optional body : String
    returns Void

  optional method sendEvent
    mandatory eventId : String
    optional body : String
    returns Void
```

The optional methods that we included in listing 5.1 are specific messages to directly send messages to the base, the measurement, and the event bus. They assume a message hierarchy as provided in figure 5.6 including their specific attributes. Therefore, the `sendMeasurement` method for example takes in a metric identifier and a value as the specific attributes of a measurement message as well as an optional message body.

Message Cache

The metric developer and metric operator need to be able to "see inside" an EMI. The monitoring and logging service of the EMI already provide a good view of the operation status of the EMI and dedicated logging information. However, sometimes they just need to see whether a certain message was send without digging through log data. Additionally, they require a mechanism to resend chunks of measurement messages in maintenance tasks such as setup of a new metric kernel or resetup of a maintained metric kernel. The Message Cache acts as a data sink on the EMDB and stores all measurement messages. It should also provide ways to see these messages and to resend chunks of them based on certain criteria. Again, checking for dedicated messages is also important for testing. A test needs to verify whether a message was send by a measurement producer from a data adapter or metric kernel.

Figure 5.10 provides an overview over the recommended internal components of a message cache. In this reference the messages from the topics are received using several dedicated message receivers. Depending on the technology, however, this can also be realized using just one receiver. The receiver(s) then delegates the storage of the data to the `ReceiveController` component. This controller transforms the data into a format that is accepted by the `Database` of the Message Cache. We recommend a document store type database technology because the data is very simple without the

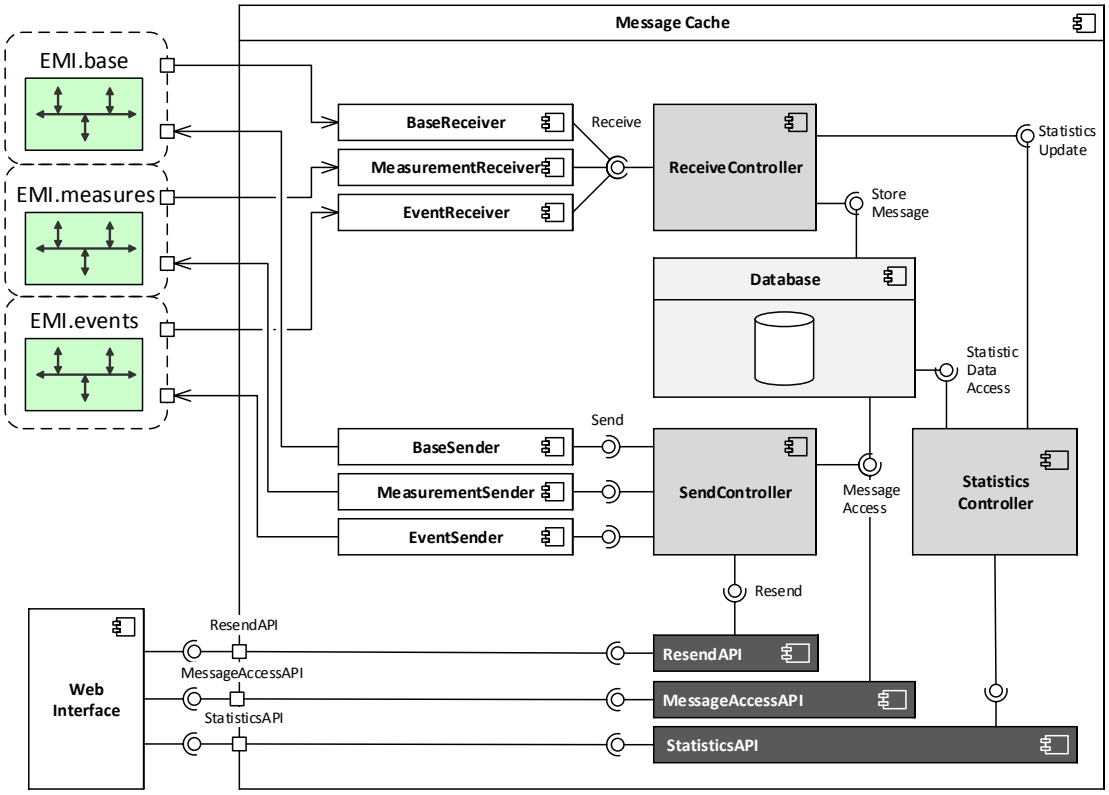


Figure 5.10.: Message Cache internal component view

need to ensure referential integrity. Therefore, storing does not require transactions and the database does not need to support foreign keys. The Message Cache provides several APIs to access it from a web user interface for example. We describe the APIs in further detail in the next paragraphs including their methods using our API description language.

ResendAPI

The core API required for maintenance. It contains several methods to resend single messages or chunks of messages. The chunks can be specified by providing a list of message identifiers or by specifying a time frame. Using this method the operator can resend all messages from a given time frame (for example the last day or week). Details on the procedures on how to use it in concrete situations are included in the operation section of our process model in section 12.2.

The `ResendAPI` delegates the implementation of the actual resending to the `SendController`. This controller queries the required messages from the database and transforms the data into messages that can be send again. The sending itself is again delegated to dedicated message sender components similar to the message gateway above.

Source Code 5.2 Resend API of the Message Cache

```
interface ResendAPI

  mandatory method resendSpecific
    mandatory messageIdentifier : String
    returns Void

  mandatory method resendTimeFrame
    mandatory from : DateTime
    mandatory until : DateTime
    returns Void
```

The method `resendSpecific` resends a specific method whereas `resendTimeFrame` resends all messages in the provided time frame.

MessageAccessAPI

Developers and operators need to quickly check if anticipated messages have been sent over the EMDB; for example when setting up an EMI or when executing integration and system tests. These tests also require access to the details of a given message to check if anticipated data is really contained in the message. Therefore the API provides a method to list the last n messages received by the message cache (for example the last 100 messages) as well as a method to access the details of a given message identifier. The following listing 5.3 provides an overview over the methods of the API using our API description language. Further details on testing are included in the design phase of our process model in section 11.2.2.

Source Code 5.3 Message Access API of the Message Cache

```
interface MessageAccessAPI

  mandatory method listAll
    optional topicIdentifier : String
    returns List of Messages

  mandatory method listLimited
    mandatory limit : Integer
    optional start : Integer
    optional topicIdentifier : String
    returns List of Messages

  optional method listTimeFrame
    mandatory from : DateTime
    mandatory until : DateTime
    optional topicIdentifier : String
    returns List of Messages
```

```
optional method listAllTopicIdentifier
  returns List of String
```

The method `listAll`, as the name suggests, returns a list of all messages in the cache. `listLimited` returns only a limited number of messages for a given limit and an optional start parameter. The optional method `listTimeFrame` can be used to show all messages in a given time frame; for example in a confirm dialog before resending them. Optionally, the message list of all methods can be filtered to only messages on a given topic using the optional `topicIdentifier` parameter. The optional method `listAllTopicIdentifiers` returns a list of all these identifiers because the topics are specific to the EMI instance and can change over time.

StatisticsAPI

This API provides access to statistics on the database required for operators. They need to be able to quickly check the amount of messages stored in the database for example to add additional storage space to the server. Additionally, because the Message Cache receives all messages on the topics the statistics provide important information on the load of the topics. This information should also be accessible from the monitoring system.

Important information includes the number of messages sent on the topic in a given time frame (for example the last hour or minute) as well as statistics like average and peak number of messages per time frame (for example average and peak number of messages per minute in the last hour).

The `StatisticsAPI` again delegates the implementation of the calculation of the statistics to the `StatisticsController`. This controller typically uses some statistics functions on the database to calculate the metrics. Additionally, it is called whenever a message is received so it may store additional data for example to calculate average and peak statistics.

This API is very specific to the needs of the operators and developers of the EMI instance. Also the access to the statistics is very specific. For example the API could just offer a single method `getStatistics` that returns a statics data transport object with all statistics. Another option would be to offer a method to access statistics specifically similar to the indicator access APIs (see section 5.3.2) using a `getStatisticValue` method that requires an identifier for the statistical value. A third option would be to offer specific methods for each statistic like `getNumberOfMessagesForTopic` or `getAverageMessagesPerMinute`. Because the options are so diverse we do not provide a specific API description for this API.

This section described the two important services in the data transport and integration layer the next section will continue with the introduction of additional, not measurement related, topics.

5.2.5. Additional Service Topics

The sections above described messages and services for the core, measurement related, topics of the EMDB. However, as introduced in section 5.2.1 the EMDB also contains additional service topics. The two most important are the monitoring topic and the logging topic.

Monitoring and Logging

The goal of the monitoring topic is to transport status informations and performance data from metric services to the monitoring service. The topic therefore integrates the (different) metric services in an EMI with the centralized monitoring system. Section 6.1 provides additional information on this integration and the information exchanged over the topic.

Similar to the monitoring topic the goal of the logging topic is to transport log data from the metric services to the central logging system. The logging topic and integration works similar to the monitoring integration. Further details on the logging system are included in section 6.2.

Command Topic

Some services in an EMI require an infrastructure to exchange asynchronous commands between each other. These commands may be required for maintenance tasks such as shutting down or halting a metric service. Commands can also be used to trigger a data adapter that implements the Invoke-Dump pattern (see section 5.4.1). As an extreme, such a command can also carry the data that should be adapted by the data adapter for example from an uploaded file (see the data adaption for CSV-files in the RiVER EMI in section 13.3 for example). The command topic provides the backbone for a loosely coupled command sender and receiver infrastructure similar to the measurement integration of the EMI.

Directory Topic

Section 6.3 describes the Directory Service. This is an optional service in an EMI that provides a number of directories to translate terms for synonyms. The section provides detailed information on the benefits of such a central directory service. This service should be easily integrateable in the metric services. Following our other designs the easiest way to integrate it is to use a dedicated topic: the directory topic.

5.2.6. Summary

This section provided details for the services and components in the data integration and transport layer of our reference architecture. The core and backbone of the layer are several topics on the EMDB following the publish/subscribe pattern. We distinguished between measurement topics to integrate metric services and service topics

for other services. The measurement topics on the EMDB exchanges EMDB Messages to integrate the data adapters and metric kernels with each other. EMDB Messages utilize generalization-specialization hierarchies to ease reuse of generic metric kernels and ease integration. This layer also contains the Generic Message Gateway and Message Cache Services. These ease development, testing, and operation of an EMI.

The message senders and receivers for the measurement topics implement the technical part of the measurement producer and measurement consumer from our formalism. Hence, the data integration and transport layer only integrates the measurement related metric services with each other (metric kernels and data adapters). However, the metric kernel also need to be integrated with the visualizations and frontend applications. This integration is realized using the calculation access layer which is described in the next section.

5.3. Calculation Access

The last section described the details of the data integration and transport layer. The enterprise measurement data bus (EMDB) in the layer implements the technical part of the measurement producers and consumers required in our formalism. This section will focus on the integration between the calculation layer and the visualization layer. The calculation layer houses a number of metric kernels which need to be accessed by the visualization frontends in the visualization layer.

On the one hand, the connection between metric kernel and visualization can be fixed. The configuration on how to access the data on the metric kernel can, thus, be hard coded into the visualization. On the other hand, if the visualizations and metric kernels should be loosely coupled then we require a mechanism to flexibly connect them with each other. The problem here, however, is that the visualizations require specific data for their visualizations. Furthermore, they need to know the metric kernels which are able to provide such data.

The idea behind our design is that the metric kernels specify the data provided by their indicators in a specific model. The visualizations can then check these models and find suitable indicators. We call this design enterprise uniform metric kernel access (EUrEKA). Even though sticking to the EUrEKA specifications is a good idea when building an EMI it is most beneficial if the metric applications require flexible association between metric kernels and visualizations or very flexible reuse and exchange of metric kernels.

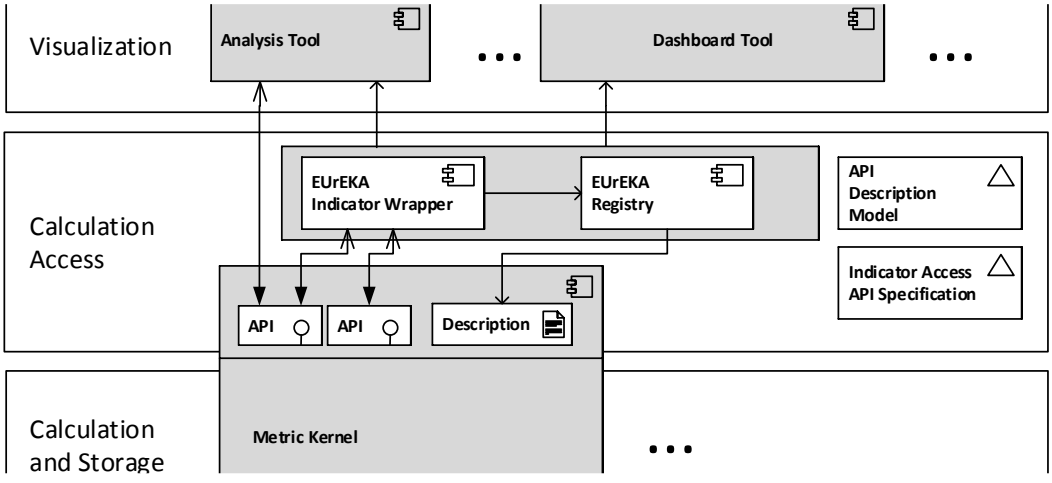


Figure 5.11.: Zoom into the calculation access layer of the MeDIC reference architecture

Figure 5.12 provides an overview over the central parts in the calculation access layer. Each metric kernel provides indicator access APIs as well as an indicator access API description. The description follows our API Description Model and the indicator access APIs implement the Indicator Access API Specification. The center shows the two

EUrEKA Services: *EUrEKA Indicator Wrapper* and *EUrEKA Registry*.

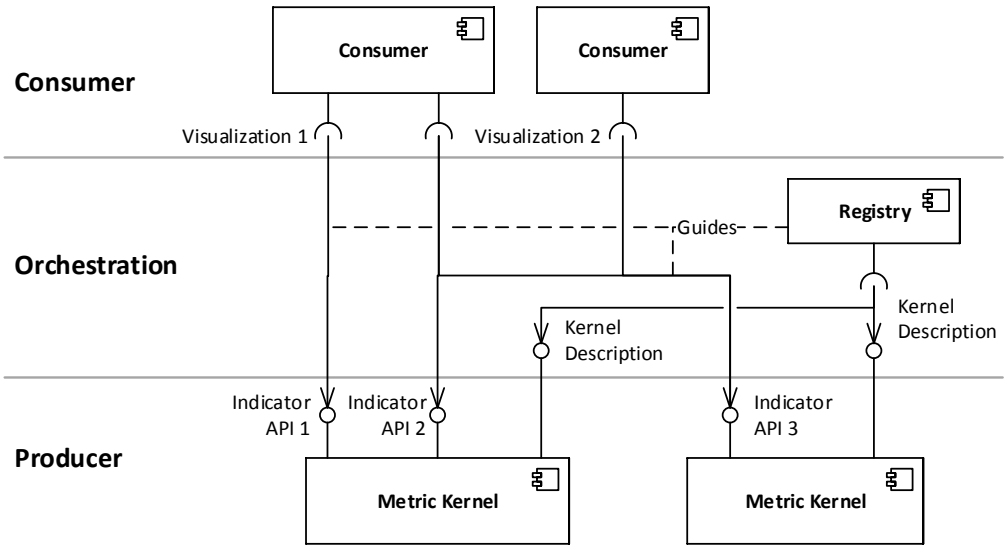


Figure 5.12.: Overview and layers of the enterprise uniform metric kernel access (EUrEKA) design

Contrasting the overview in figure 5.11, figure 5.12 shows a conceptual overview over the EUrEKA design. In EUrEKA we differentiate between three layers: the EUrEKA consumers on the top (in the EMI the visualization layer), the EUrEKA orchestration layer in the middle, and the EUrEKA providers on the bottom. The core of EUrEKA is the definition for the indicator access APIs on the EUrEKA providers (the metric kernels) as well as an API and a model to describe these APIs to the EUrEKA registry in the orchestration layer. The indicator access APIs, as the name suggests, allow access to the indicator values of the metric kernel following the REST principle. The figure shows three different indicator access APIs on the two metric kernels. These APIs are registered in the EUrEKA registry. EUrEKA consumers (visualizations) can query the EUrEKA registry to get a list of suitable indicators on metric kernels that fit their data needs. For example the left consumer in the figure requires specific data for visualization 1 which is provided by Indicator Access API 1. The two consumers share a visualization (Visualization 2) which can visualize data either from Indicator API 2 or Indicator API 3 (these two APIs both provide compatible data for visualization 2).

The following subsection 5.3.1 will briefly discuss our design decisions for EUrEKA and provide a quick discussion on related work such as UDDI and OData. We will then introduce the EUrEKA indicator access API description model in subsection 5.3.3. Instances of this model describe the indicator access APIs on the metric kernels. Subsection 5.3.2 will describe these indicator access APIs using our API description language. From there we will introduce the reference architectures for the two services in this layer. We start in subsection 5.3.4 with the reference architecture for the EUrEKA Registry Service. Subsection 5.3.5 will provide details about the optional EUrEKA

Producer Gateway. The next subsection 5.18 provides additional information about the EUrEKA Consumer Component which is used in the visualization frontends. After that we introduce the reference architecture for the optional EUrEKA Indicator Wrapper Service in subsection 5.3.7. We close the section with a short summary in subsection 5.3.8.

5.3.1. Design Decisions and Related Work

Contrasting the data transport and integration layer, the communication in the calculation access layer is purely based on interactions between REST service. We decided to use this type of integration infrastructure in this layer because the control flow in this upper part of an EMI is triggered by interactions on the frontend applications in the visualization layer. Thus, inversion of control resulting from the push-based mechanisms in the EMDB using topics is not suitable here. Therefore, it does not require a bus-based publish/subscribe infrastructure. However, depending on the requirements of the EMI that should be build parts of it can be implemented using an enterprise service bus product.

Integrating and discovering the different metric kernels in an EMI is challenging. Service oriented architectures (SOA) struggle with a similar yet more general problem[2], for some time. The solution for this problem in generic SOAs is Universal Discovery, Description and Integration (UDDI). UDDI includes a directory of all services in a SOA as an XML-based registry. The registry contains descriptions of the services using their WSDL descriptions. Event though UDDI was quiet broadly adopted in the early 2000 years it failed in the long run. Starting in 2006 Microsoft, SAP, and IBM discontinue their UDDI service support [RMGW05]. In 2010 Microsoft removed UDDI services from Windows Server 2008 [Net10]. We believe the reason why the UDDI initiatives failed are because they tried to solve the integration and discovery problem on a too general level. Arbitrary SOA services are to different to just use a uniform registry and generic integration mechanism for all of them.

Consequently, alternative solutions and services started to take over the role and idea of UDDI. One of these alternatives is the open data protocol (OData - see `http://www.odata.org/`). OData enables services, similar to UDDI, to define their services and their data models. Contrasting UDDI, OData uses REST services and specific URIs for the location of the metadata information and service integration. However, the goal of OData, like UDDI, is to integrate arbitrary services with each other. Another alternative that is focused on load balancing is Netflix Eureka (see `https://github.com/netflix/eureka` for further details). The backbone of Netflix Eureka also provides a service registry. Its main porposes are middle tier load balancing and service location. The concept of Eureka is that the service registry provides the load balancer with services suitable for a specific task, which eases and automates its configuration. Consumers simply request resources from the load balancers who delegates the calls to the actual resources (in the cloud). Compared to traditional load balancers this does not require to hard-code the service URIs into the load balancer, which is particularly important for

[2] In an arbitrary SOA the services are arbitrary. However, in our case we know that all the APIs are indicator access APIs. This is a fixed semantic and we also ensure technical compatibility

cloud services whose URIs change frequently.

5.3.2. EUrEKA Indicator Access APIs

The primary goal of the indicator access APIs is to access the result of metric calculations as indicators. Indicators are calculation results processed for visualizations (see section 5.1.1). Consequently every calculation result that should be visualized needs to be accessible via one of the indicator access APIs of the metric kernel. A metric kernel can provide as many indicator access APIs as the design requires. It can also provide several slightly different indicator access APIs for the same calculation result if it needs to be visualized in different ways and the visualizations require other types of data. Following our formalism, the calculation of the metrics can be tailored by providing a variability configuration that fits the variability model of that metric. Therefore, the configuration needs to be passed as parameter[3].

The indicator access APIs can also contain methods to increase the usability of the interactions in the visualizations. They can providing information about metrics and about calculation data insight the metric kernel; like a list of all metric identifiers that are implemented in the metric kernel. Furthermore, the API contains a method to indicate the data quality of a specific indicator request. This information can be visualized alongside the visualization to provide information about data quality problems. An alternative to this design would be to always include the data quality in a special wrapper object with each calculation result. However, this would produce additional data overhead if the data is not used in the visualization. Also, if the indicator data is included in a wrapper alongside the data quality then the data type of the method would differ from the type provided in the API description model.

The following section contains additional information and a detailed description of the required API (EUrEKA indicator access API).

Indicator Access APIs

Indicator access APIs most importantly require methods to access data (the indicators). We differentiate between a method to access the current data (latest value of a calculation) and a method to access a list of data (for example a time series of calculations). These two types could also be differentiated via different metric identifiers but this would lead too unnecessary many metric identifiers. Another mandatory method on the APIs is a method to access the data quality for a given metric and a given entity of measurement. All other methods that can be used to increase the usability of the specification of monitors are optional.

The following listing 5.4 specifies the indicator access APIs using our API specification language (see section 3.2 for details on the language). We describe each method in detail after the listing.

[3]See thesis of Martin Mädler for further details [Mäd12].

Source Code 5.4 Metric Value Access API specification

```
interface MetricKernelCalculationDataAPI
  generic type parameter DataType

  mandatory method getCurrentDataValue
    mandatory parameter entityOfMeasurement : String
    mandatory parameter metricId : String
    optional  parameter variabilityConfig : VariabilityConfiguration
    returns DataType

  mandatory method getDataSeries
    mandatory parameter entityOfMeasurement : String
    mandatory parameter metricId : String
    optional  parameter variabilityConfig : VariabilityConfiguration
    returns List of DataType

  mandatory method getDataQuality
    mandatory parameter entityOfMeasurement : String
    mandatory parameter metricId : String
    returns DataQuality

  optional method getVariabilityModelForMetricId
    mandatory parameter metricId : String
    returns VariabilityModel

  optional method getAllMetricIds
    returns List of String

  optional method getAllEOMsForMetricId
    mandatory parameter metricId : String
    returns List of String
```

getCurrentDataValue Returns the current value for a given entity of measurement and metric reference. For example this method can be used to access the current calculation result of a metric in a list or on a bullet graph. It is important to differentiate between *current* and *latest* value of the metric. The latest value is simply the calculation result for the data with the highest time stamp. This can be in the future and is typically not the value required in a dashboard. The current value is the value suitable for the time of the access which depends on the specification of the metric.

getDataSeries Returns a list of data values for a given metric reference and entity of measurement. For example this method can be used to access a time series of the calculation results of a metric.

getDataQuality Returns the current data quality for a given entity of measurement and metric reference. The resulting data quality type needs to be specified during the design metric services and integration activity in the specification phase (see 11.2.1). Obviously, the visualization needs to be able to visualize the data quality if required.

getVariabilityModelForMetricReference Returns an instance of a variability model for the given metric reference. This variability model can then be used to create specific editors to configure variability configurations when specifying data sources for monitors.

getAllMetricIds Returns a list of all metric references provided by this kernel for this specific data type. This is a convenience method to increase the usability of the visualization frontends. The list can be used to provide a selection of metric ids in a user interface when specifying data sources for monitors.

getAllEOMsForMetricReference Returns a list of all eoms that are stored for a specific metric reference. This also is a convenience method for the configuration of the visualization because it allows to reduce the eom selections to only those that provide calculation results for the given metric reference.

5.3.3. EUrEKA Kernel Description Meta Model

As stated above, the metric kernels need to describe the indicator access APIs and the data on these APIs to allow a visualization to check this data against its requirements. The EUrEKA Kernel Description Meta Model defines the models that describe the indicator access APIs of a metric kernel and their data. These descriptions, the models, can be stored in the EUrEKA registry and can be checked by the visualizations. The meta model itself is very similar to the web service description language (WSDL). Our meta model, however, is more specific and less universal then WSDL. Therefore it contains less overhead and models are easier to specify, to read, and to store.

In the meta model, we choose to reference the types in the models only implicitly using strings and not explicit using object references because this makes the model more flexible and the resulting models do not need to specify the simple types again (they are used implicitly). This makes the models more compact and thus easier to read. Furthermore, we can include keywords in the string such as `list of` and `set of` to indicate lists and sets of data without the need to model these explicitly in the meta model. This again makes the meta model easier to understand and the models easier to read because they are more compact.

Figure 5.13 provides a UML class diagram for the EUrEKA metric kernel description meta model. The root of the model is the `Metric Kernel` itself. The metric kernels are identified by their name and can provide an additional description. The metric kernel contains a number of indicator APIs. Each `Indicator API` contains the (absolute) URL to the API as well as the data type that is used as value for the generic type parameter for the API (see section 5.3.2). The metric kernel also needs to define the data types used in the description. Each `Data Type` contains of a name and a type. We

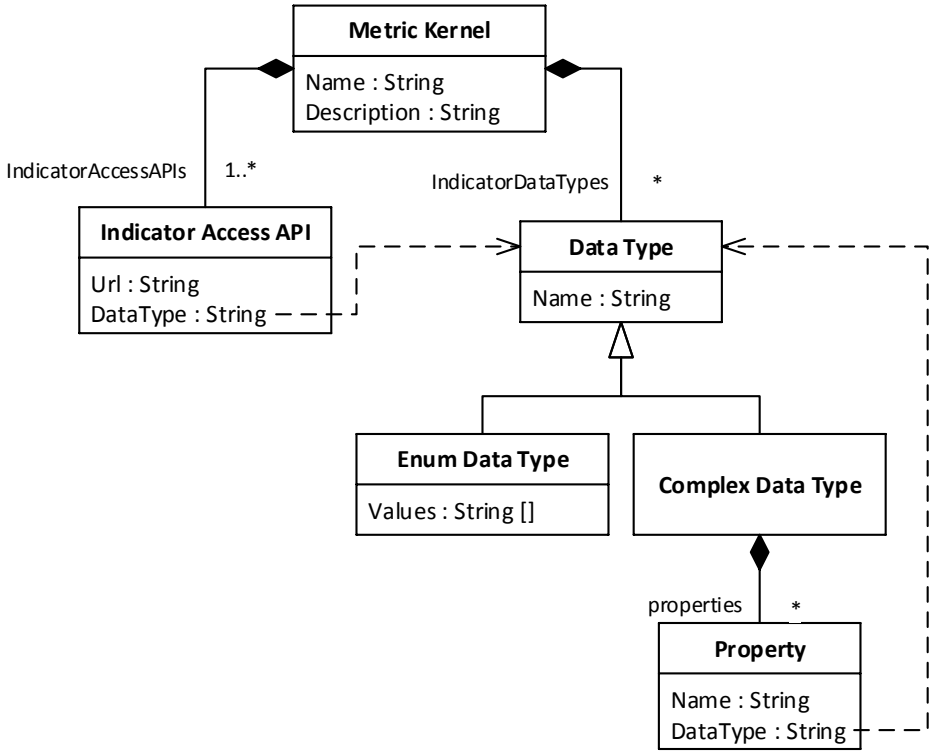


Figure 5.13.: EUrEKA metric kernel description meta model as UML class diagram

differentiate between three different types: `simple`, `complex` and `enum`. Simple data types are implicit and don't need to be formally specified. We differentiate between the primitive data types: `string`, `double`, `integer`, `boolean`, and `date`. These data types can be extended if required in the specific EMI. A `Complex Data Type` includes a list of properties. Each `Property` contains of name and a type. The type needs to refer to another data type specified with the metric kernel or a primitive type. The `Enum Data Type` contains a list of enumeration values.

EUrEKA Kernel Description Model Example

The following listing 5.5 provides an example for a metric kernel description using the JSON data format. Contrasting the meta model, this JSON description also requires a type attribute for each meta model data type to differentiate between the different types when demarshalling the models. The kernel described in the listing provides two indicator access APIs to access different indicators for risk metrics.

Source Code 5.5 Example of a metric kernel description using JSON

```
{ name: "Risk Metric Kernel",
  description: "The risk metric kernel provides a number of
    indicators to analyse project risks. The indicators include
    statistical indicators for metrics like number-of-open-risks
    as well as an indicator for the current risks of a project."
  indicatorAccessAPIs : [
    { url: "http://localEMI:8080/emi.risks/rest/categMeasures.json",
      dataType: "categorizedMeasures"
    },
    { url: "http://localEMI:8080/emi.risks/rest/risks.json",
      dataType: "set of risk"
    }
  ],
  indicatorDataTypes : [
    { type: "complex",
      name: "categorizedMeasure",
      properties: [
        {name: "category", type: "string"},
        {name: "value", type: "double"}
      ]
    },
    { type: "complex",
      name: "risk",
      properties: [
        {name: "identifier", type: "string"},
        {name: "probability", type: "string"},
        {name: "impact", type: "string"},
        {name: "status", type: "string"},
        {name: "development", type: "riskDevelopment"}
      ]
    },
    { type: "enum",
      name: "riskDevelopment",
      values: [
        "increase",
        "noChange",
        "decrease"
      ]
    }
  ]
}
```

The first indicator access API of the metric kernel is located at the URL `http://localEMI:8080/emi.risks/rest/categorizedMeasures.json`. The generic data type parameter of this API is set to the data type `categoriezedMeasures`. The data type description defines this data type as a complex data type which contains of the two properties category and value. The type

of category is `string` and the type of value is `double`. This is a typical data type that can be visualized using a bar or line chart.

The second indicator access API of the metric kernel is located at the URL `http://localEMI:8080/emi.risks/rest/risks.json`. The generic data type parameter for this API is set to the data type `set of risk`. Therefore the API will return sets of the `risk` data type. The data type description defines the `risk` data type as a complex type with the properties: identifier, probability, impact, status, and development. Except the development property all other properties are of type `string`. The development property is of type `riskDevelopment` that is defined below as and enum with the values: increase, noChange, and decrease. The values from this API can be used to feed a risk matrix that provides a quick overview over all the risks of a project.

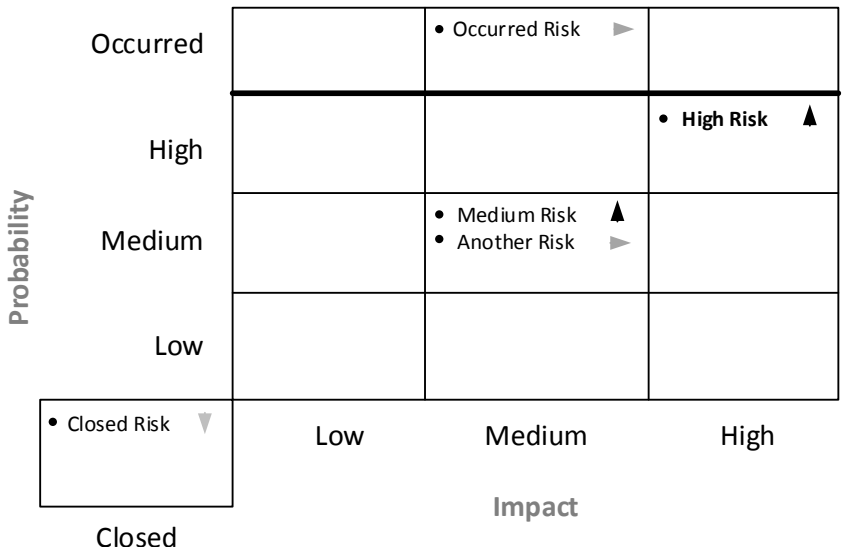


Figure 5.14.: Example for a risk matrix that can be feed by the risk data type from the previous example

The development property allows to also visualize the change of a specific risk in this matrix as shown in figure 5.14. The Matrix shows the impact of the risk on the x-axis and the probability of the risk on the y-axis. Contrasting traditional the traditional risk matrix designs we added *Occurred* as well as *Closed* to the probability to get a better picture of the project risks. Each compartment contains the risks with this specific set of impact and probability values. The arrows next to each risk indicate the development of that specific risk. The risks *Another Risk* and *Occurred Risk* remained stable (did not change) whereas the *High Risk* and *Medium Risk* increased. The *Closed Risk* decreased down to the closed container.

5.3.4. EUrEKA Registry

The EUrEKA Registry Service provides the orchestration backbone of the EUrEKA reference architecture. It provides visualizations with a way to find suitable indicators to feed them data without the need for the visualization to know each metric kernel in an EMI. Therefore, this service helps to decouple the visualizations and the metric kernels from each other. This simplifies their design, increases their reuseability, and eases the operation of the EMI, which all reduces maintenance effort.

The EUrEKA registry service is not mandatory for an EMI but the alternative is to hard code metric kernel to visualization mappings and indicator access APIs. Alternatively, the routing and access between visualizations and metric kernels can be realized using an of-the-shelf enterprise service bus (ESB) product to put the configuration in one place and don't hardcode it into the visualization. However, this increases the structural and the configuration complexity of the EMI. Furthermore, it requires additional knowledge for the configuration of the ESB and its components.

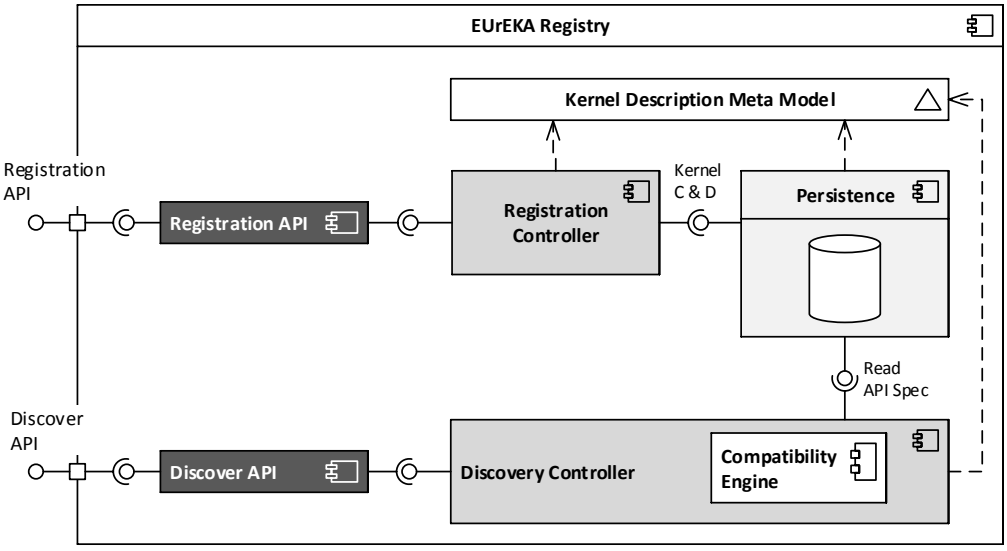


Figure 5.15.: EUrEKA registry service internal component view

Figure 5.15 provides an overview over the recommended internal components of the EUrEKA Registry Service. The service is, as always in the reference architecture, designed as a microservice. Hence, the service only provides very limited functionality. The internal structure is, therefore, very simple as well. The foundation of all the components is an implementation of the `Metric Kernel Description Meta Model` described above. Kernel description models are used for the discovery mechanism and hence stored in the database of the service. The `Registration Controller` allows to store and delete metric kernel description models using the `Registration API`. These models are then accessed by the `Discovery Controller`. This controller uses its `Compatibility Engine` to discover indicator access APIs in metric kernel description

models that are compatible to the data type required by a visualization. This functionality can be accessed via the `Discover API`. The following two subsections use our API specification language (see section 3.2) to specify the APIs in greater detail.

Registration API

The goal of this API is to register and unregister metric kernels with the EUrEKA Registry Service. The API can either be accessed directly by the metric kernels so they can register themselves once they startup or they are used with a small web UI so the registration and unregistration can be done by an operator. Obviously, only registered metric kernels are discovered. We believe the method names are intuitive and we do not need to provide further descriptions.

Source Code 5.6 Metric Kernels Registration API specification on EUrEKA Registry

```
interface RegisterKernelAPI

  mandatory method registerMetricKernel
    mandatory parameter metricKernelDescriptionURL : String

  mandatory method unregisterMetricKernel
    mandatory parameter metricKernelDescriptionURL : String
```

Discover API

The goal of this API is to allow a EUrEKA consumer (visualization) to receive a list of Indicator Access APIs which provide data that is compatible with a given data type. The compatibility definition follows our definition for compatibility in our formalism in section 2.3.5. In short: An indicator access API is compatible to a given data type if for each property in the given type there exists a compatible property in the data type returned by the API. However, we need to extend this definition for the enumeration data types. An enumeration d_1 is compatible to another enumeration d_2 if d_2 is embedded in d_1. An enumeration d_2 is embedded in d_1 iff for all enumeration values of d_2 there exists an equal enumeration value in d_1.

Source Code 5.7 Metric Kernel Discover API specification on EUrEKA Registry

```
interface DiscoverAPI

  mandatory method discoverCompatibleAPIs
    mandatory requestedDataType : DataType
    returns List of String // List of Indicator Access API URLs

  mandatory method getKernelNameForIndicatorURL
    mandatory indicatorAccessAPIURL : String
    return String // Name of metric kernel that provides the API
```

The `discoverKernel` method searches trough the list of all registered metric kernels and checks if they provide a data type that matches the required data type. If they do then the URL of the Metric Value Access API is added to the result list. The registry services utilizes the compatibility engine component to implement the check. The engine can either be implemented using a recursive algorithm or it can use a number of compatibility-check objects that are instantiated based on the structure of the required data.

The `getKernelNameForIndicatorURL` method, as the name suggests, return the name of the metric kernel that provides the indicator access API at the given URL. This method performs a reverse search in the persistence of the registry service to retrieve the kernel name. This method is, for example, used by the EUrEKA consumer component to check the operation state of a metric kernel[4] before accessing the indicator access API.

5.3.5. EUrEKA Producer Gateway (optional)

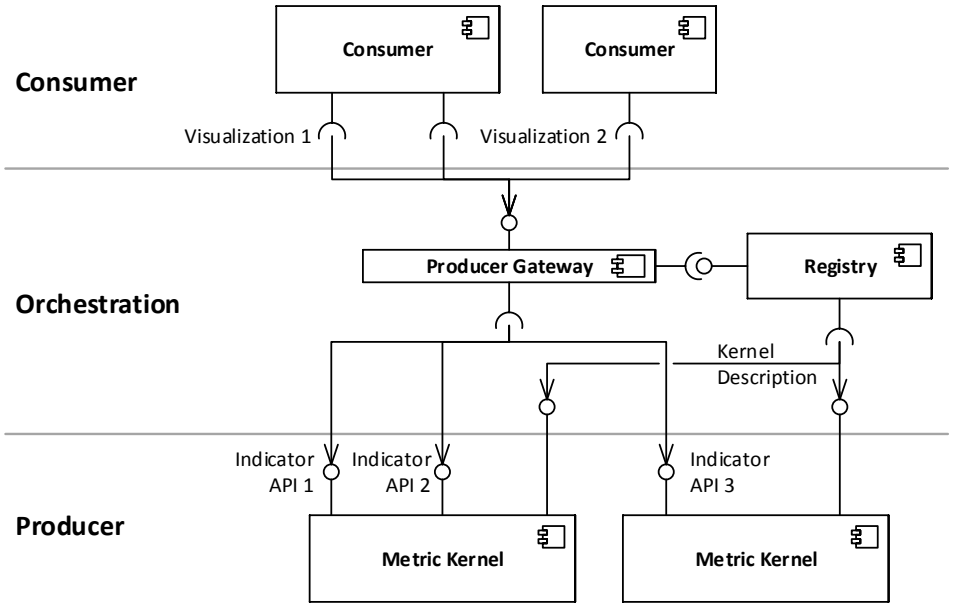


Figure 5.16.: EUrEKA overview using the EUrEKA producer gateway service

Without the EUrEKA Producer Gateway Service each EUrEKA consumer (visualization) needs to store the actual URLs from the indicator access APIs in their configurations to access their data. This is a hard coupling between the EUrEKA producers and EUrEKA consumers, at least to a certain degree. This coupling can become problematic if, for example due to performance problems, a metric kernel needs

[4]The operation state can be retrieved from the monitoring service using the name of the EMI service. The EUrEKA consumer component, however, provides access to indicator access APIs. Hence, before accessing the API it needs to lookup the actual kernel name that implements the provided indicator access API.

to be relocated to another server, which changes the URLs because of a new physical location. Then all the configurations in the EUrEKA consumers that require data from one of the indicator access APIs of this kernel need to be updated which is time consuming and can lead to errors.

The EUrEKA Producer Gateway provides a central component that decouples the EUrEKA consumers and EUrEKA producers from one another. Figure 5.16 shows an overview over EUrEKA including a EUrEKA Producer Gateway Service. Compared to the version without the gateway in figure 5.12 all calls from the consumer layer now pass through the gateway and do not connect directly to the producers anymore.

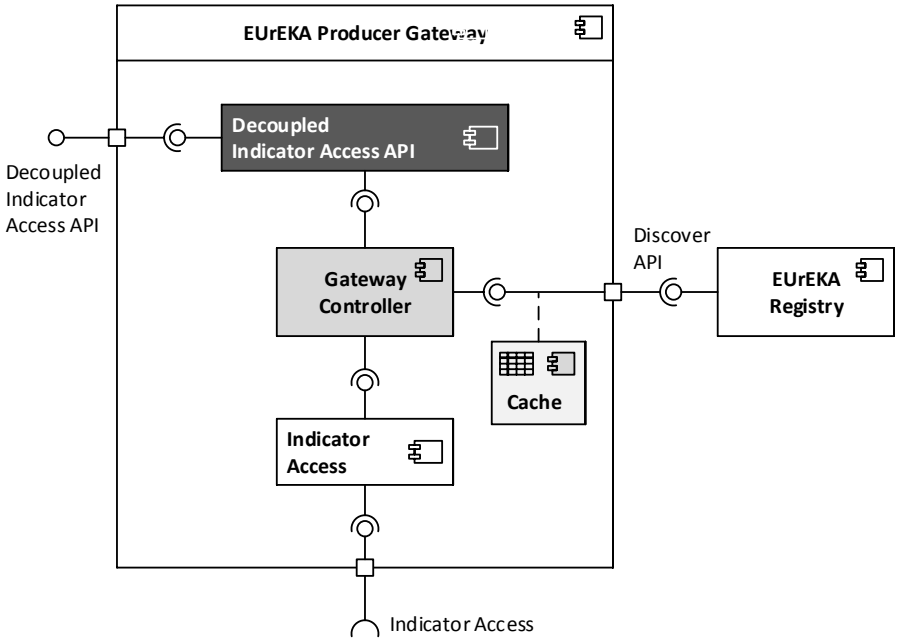


Figure 5.17.: EUrEKA producer gateway service internal component view

Figure 5.17 provides an overview over the recommended internal components of the EUrEKA Producer Gateway service. The EUrEKA producer gateway provides a decoupled indicator access API that delegates calls to the actual indicator access APIs of the EUrEKA producers. The gateway just delegates the indicator requests. Hence it does not use a EUrEKA consumer component and does not perform maintenance status checks. The maintenance status check needs to be performed before calling the EUrEKA producer gateway by the consumers on top. The identification of the EUrEKA producers uses the name of the producer (metric kernel) and the data type. Using this the EUrEKA producer gateway can use the EUrEKA registry to get the URL of the actual indicator access API, which can be locally cached to increase performance[5].

[5]The local caching will, however, require to constantly synchronize the cache with the registry. This adds additional communication overhead and complexity to the registry. Yet, the registry is rarely updates. Hence, the overhead is not to high.

We believe, the design of the gateway is easy to understand . Hence, we do not provide additional specifications for the APIs or the dynamic behavior of the service.

Every access from the visualization layer to data in the calculation layer needs to go through it. Therefore the gateway can become a bottle neck for the data flow from the EUrEKA producers to the EUrEKA consumers. However, this can be addressed by providing sufficient resources to the service and potentially using multiple producer gateways in parallel with a central load balancer on top[6]. However, a nice side effect of using the gateway is that it can easily provide statistical information on the usage and load of communication in the calculation access layer of the EMI, which is an important performance indicator for monitoring.

When instantiating an EMI from this reference architecture the architects need to decide if they want to include this service or not based on the needs and scaling of the actual EMI.

5.3.6. EUrEKA Consumer

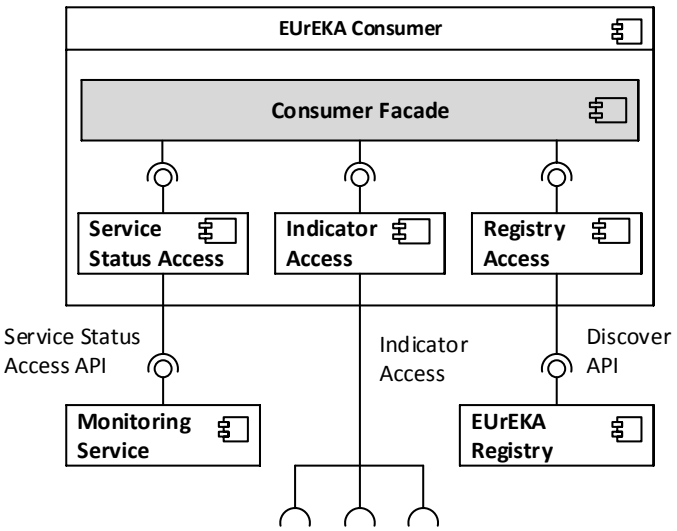


Figure 5.18.: EUrEKA consumer internal component view

The EUrEKA consumer component provides the consumers of indicators with a ready to use component to access the indicator access APIs as well as access to the EUrEKA registry. For example this component is used by all the visualization frontends of an EMI. Hence, this component is typically provided by the EMI development platform.

The EUrEKA consumer houses components to access the EUrEKA indicator access APIs from the EUrEKA producers and performs additional checks before accessing the APIs. Therefore, each EUrEKA consumer also connects to the service status access API of the monitoring service and the EUrEKA registry service. The service status

[6]This could easily be implemented using Netflix Eureka

API of the monitoring service provides status information about all services in an EMI. It is important to check the service status of the metric kernel before accessing the indicator access API of the metric kernel because the metric kernel may be in a non functional operation state like *offline* or *maintenance*. If it is being maintained then the call to the indicator access API will most likely fail or produces wrong or outdated results. Furthermore, the service status API can provide additional information about the operation state. This information can be presented to the metric customers; for example the anticipated maintenance time. See section 6.1 for further details on the service status API.

5.3.7. EUrEKA Indicator Wrapper (optional)

Typically visualizations can and need to visualize multiple indicators (like Cartesian charts which can visualize multiple bar or line series). Instead of configuring the indicator access in the visualization or monitor configuration we find it more useful to delegate this configuration to a separate service. Therefore, the EUrEKA Indicator Wrapper Service provides a single point of access to wrapped data from multiple indicator access APIs from multiple metric kernels. This service is optionally and the architect of an EMI instance can choose not to include it and configure this somewhere else. However, as EMIs get larger with more metric kernels, more visualizations, and more complex configurations the need and usefulness for this service will raise.

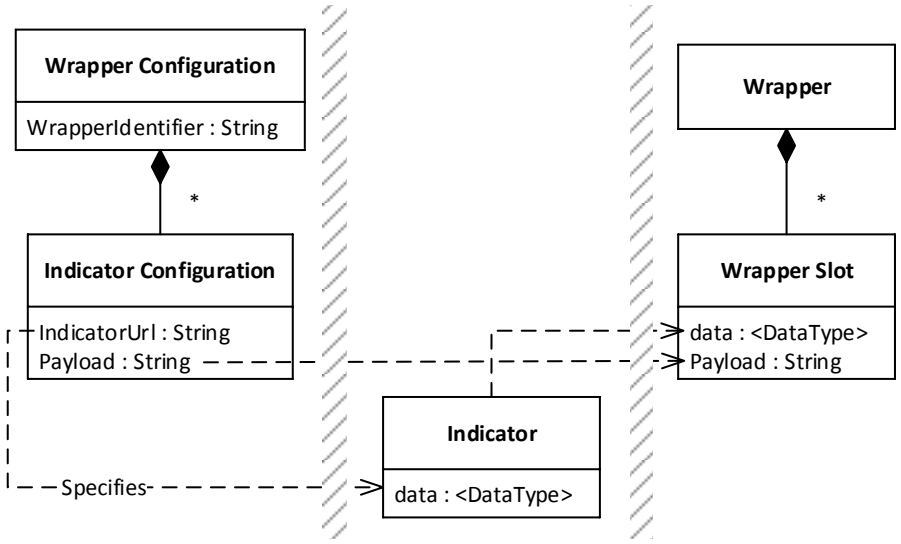


Figure 5.19.: Wrapper Configuration Model and Wrapper production

Figure 5.19 provides a UML class diagram of the model for wrapper configurations and an overview over the production of the wrappers from the indicator data. A `Wrapper Configuration` is identified by its Wrapper Configuration Identifier. The configuration itself is just a container for `Indicator Configurations`.

Each indicator configuration contains the URL for the actual indicator or when using a EUrEKA producer gateway (see section 5.3.5 above) the name of the metric kernel and the data type. Additionally, indicator configurations provide an additional payload that is added to the wrapper slot when wrapping the data from the indicator. This payload can be used to store specific configuration from the visualizations for the data of this particular indicator. The visualization can then use this payload to change the rendering of this data. For example this provides an easy way to specify the visualization type (bar or line) of a Cartesian chart.

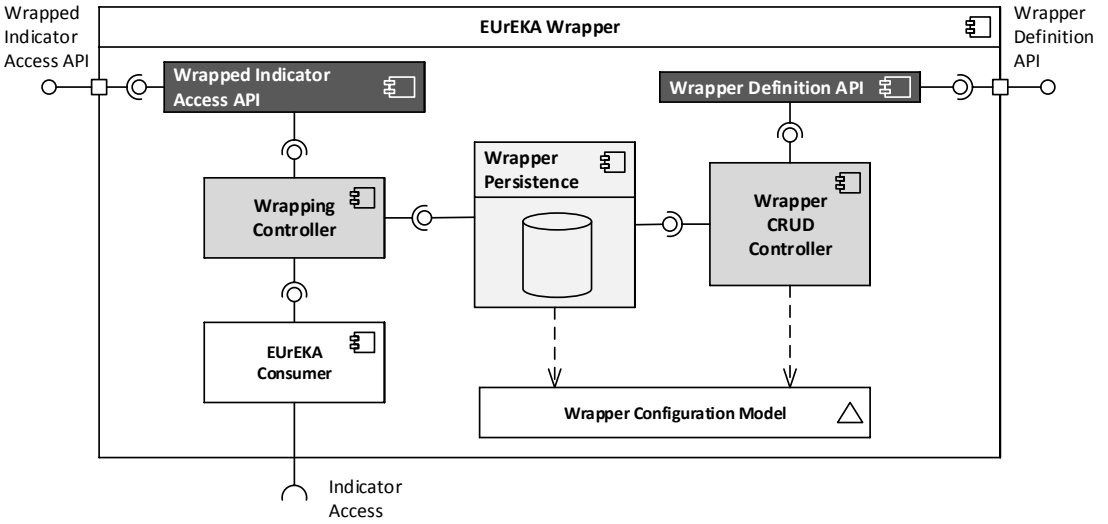


Figure 5.20.: EUrEKA indicator wrapper service internal component view

Figure 5.20 provides an overview over the recommended internal components of the EUrEKA Indicator Wrapper Service. The wrapper service uses a `EUrEKA Consumer` component to access indicator access the data from the indicator access APIs configured in a wrapper configurations. The actual wrapping and access to the indicators is orchestrated and controlled by the `Wrapper Controller` component which accesses the wrapper configurations from the `Wrapper Persistence` component. The wrapper service provides two APIs for the communication from the visualization layer. The `Wrapper Definition API` for the definition, manipulation, and deletion of wrapper configurations as well as the `Wrapper Indicator Access API` to access the wrapped data. The following two subsections use our API specification language (see section 3.2) to specify the APIs in greater detail.

Wrapper Definition API

This API provides methods to create, update, and delete wrapper configurations. Optionally it can provide methods to clone wrapper configurations to ease their definition.

Source Code 5.8 Wrapper Definition API specification on the EUrEKA Indicator Wrapper

```
interface WrapperDefinitionAPI

  mandatory method createWrapperConfiguration
    mandatory wrapper : WrapperConfiguration
    returns Void

  mandatory method updateWrapperConfiguration
    mandatory wrapperIdentification : String
    mandatory wrapper : WrapperConfiguration
    returns Void

  mandatory method deleteWrapperConfiguration
    mandatory wrapperIdentification : String
    returns Void

  optional method cloneWrapperConfiguration
    mandatory wrapperIdentificationFrom : String
    mandatory wrapperIdentificationClone : String
    returns Void
```

We believe the method names are intuitive and we do not need a further description for them.

Wrapper Indicator Access API

This is simply an instance of our indicator access API specified above in section 5.3.2. The data type of this API, however, is `Indicator Wrapper` and not a specific data type from the calculation layer. More specifically, the data type of a concrete call is `Wrapper of <DataType>` for the actual data types returned by the indicator access APIs configured in the concrete wrapper configuration. Therefore, the indicator access API of the wrapper service is a generic indicator access API because it provides multiple (generic) data types.

5.3.8. Summary

The last section introduced the internal details and components in the calculation access layer of our technical reference architecture. The core is the specification of the indicator access APIs on the metric kernels together with a metric kernel description meta model and a registry service. The registry service provides mechanisms to discover indicator access APIs based on a data type required for a visualization. The information about the APIs as well as their data types is provided by metric kernel description models. Because the API follow the specification for indicator access APIs the visualization are able to uniformly access them regardless of their actual data types. Furthermore we specified optional additional services that ease the interaction and solve specific problems with the indicator access APIs on the metric kernels.

Contrasting the Enterprise Measurement Data Bus (EMDB) the design for the Enterprise Uniform Metric Kernel Access (EUrEKA) does not utilize a messaging infrastructure. The main reason for this is the opposite control flow in EUrEKA as well as performance optimization to provide fast access to the calculation results. A EUrEKA infrastructure is easily set up using existing tools such as Netflix Eureka (which even shares the name but is aimed at something else), OData with Apache Olingo, or Spring Cloud or the services can be build from scratch. All services are again designed as microservices multiple different technologies and existing services and tools can be used to build the EUrEKA infrastructure for an EMI instance. Furthermore, the last section also included reference architectures for each of the services if they should be build from scratch.

This wraps up the description of the two integration and communication layers of our technical reference architecture. The following sections will address the reference architectures for the EMI specific components in the three domain layers. We start with the reference architecture for data adapters in the bottom layer of the reference architecture.

5.4. Data Adapter Reference Architecture

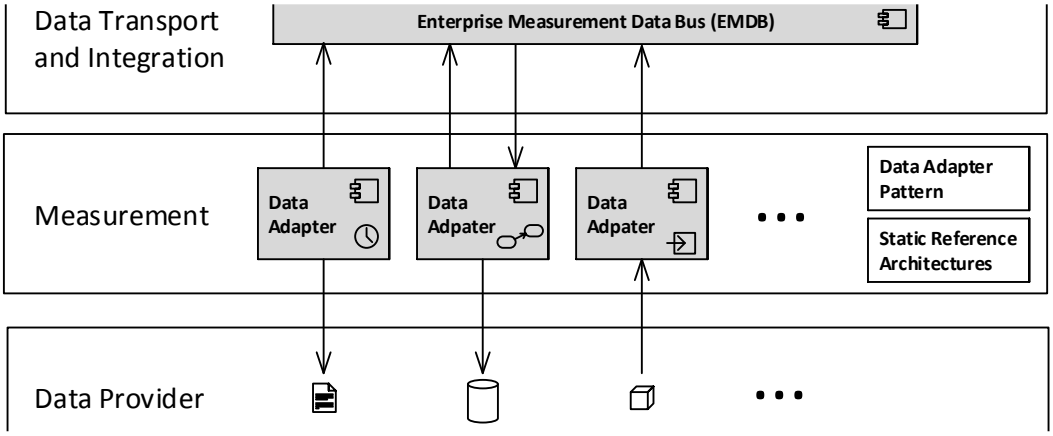


Figure 5.21.: Zoom into the data adapter layer of the MeDIC reference architecture

The last two sections focused on the two communication layers: Data Transport and Calculation Access. This section will now focus on the first functional layer of our reference architecture. Figure 5.21 shows an overview over the data adapter layer. Contrasting the last sections, the reference architecture for this layer does not contain any fixed services. The reference architecture provides a variety of patterns for the adaption of the data from the data adapters together with specific reference architectures for each of these patterns. These pattern guide the design of the actual data adapters when designing an EMI and specifying the data adapter.

The following subsection 5.4.1 provides the details on the four adapter patterns that we identified when building the EMIs for our field studies. After that, subsection 5.4.2 will provide the static reference architectures that guide the implementation of a data adapter based on the adapter patterns.

5.4.1. Adaption Patterns and Dynamic View

Figure 5.22.: Icons for the different data adapter pattern

The heterogeneity of the systems that need to be integrated in an enterprise measurement infrastructure calls for flexible data adaption mechanisms (requirement ReD-IM1). Hence, the reference architecture provides four different adaption patterns: *Push-Forward*, *Pull-Forward*, *Invoke-Pull*, and *Invoke-Dump*. Figure 5.22 shows the

different icons that we use to indicate the type of data adapter pattern in component or overview diagrams. We describe the concepts and application scenarios of each of these adapter pattern in the following subsections.

Push-Forward

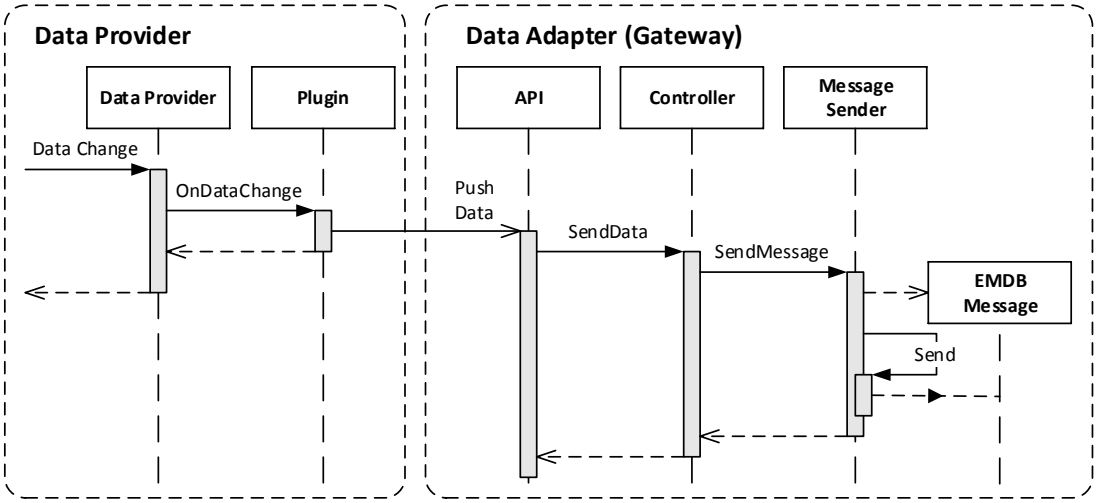


Figure 5.23.: UML sequence diagram for the concept of the push-forward adapter pattern

The idea of the pattern is that changed data in the data provider is pushed to the EMI immediately after the change. Hence, the Push-Forward data adapter pattern guarantees the best latency between change event in the adapted system and the visualization, which satisfies requirement ReD-IM2. The sequence diagram in figure 5.23 shows the interaction of the different parts of the push-forward adapter pattern as UML sequence diagram. The adaption is triggered by a data change in the data provider. This change then triggers a plugin in the data adapter that is registered to be called on data change. This plugin just calls an API on the data adapter in the EMI with the specific data that was changed. Because the plugin is not located in the EMI this satisfies requirement ReD-IM3. This data adapter then creates a specific EMDB message for the data and sends the message to the EMDB. The data is then transported to the metric kernels and the measurement cache. Hence, the visualization components could immediately update the visualizations to reflect the new data.

The data adapter in the EMI that provides the API, which the plugin calls, only transforms the data and generates an EMDB message. Therefore, these data adapter are called *Data Gateways*.

Pull-Forward

Standard BI (Business Intelligence) systems use scheduled jobs[7] to adapt the data from a data provider. The Pull-Forward data adaption pattern is inspired by these ETL jobs. Figure 5.24 shows the interaction of the different parts of the pull-forward adapter pattern as UML sequence diagram. The extract task is triggered by a scheduler who is configured to a certain interval like every minute, hour, or day. The extract task then retrieves the changed data from the data provider using an API on the data provider. After that, the data adapter needs to loop over all the changed data in order to pack it into EMDB Messages and send them to the EMDB.

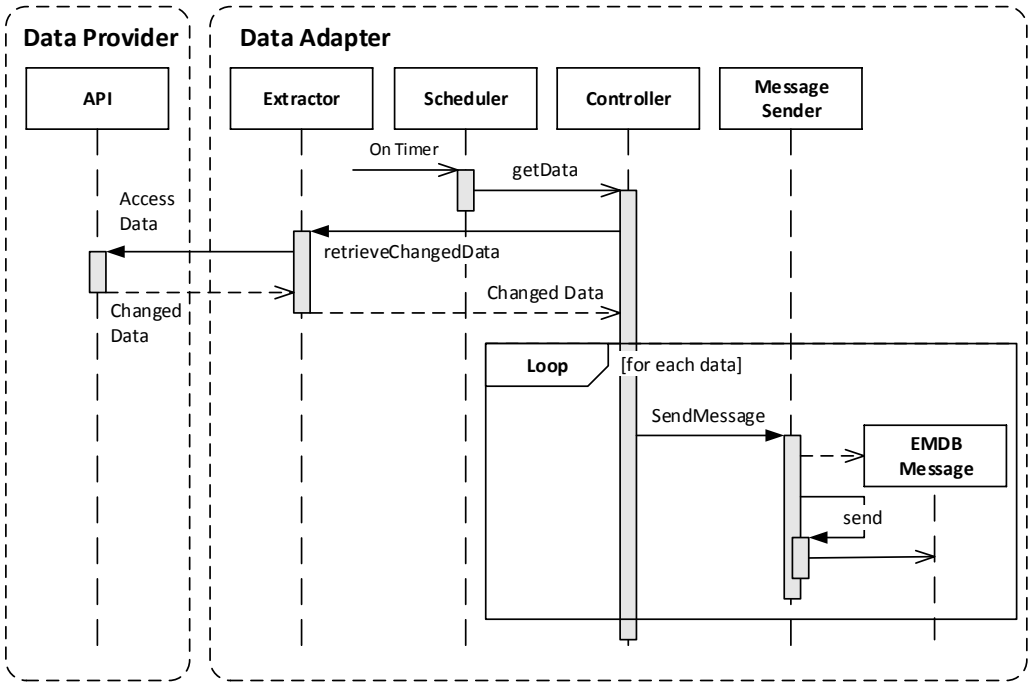


Figure 5.24.: UML sequence diagram for the concept of the pull-forward adapter pattern

Even though this adapter pattern is inspired by the most popular way to get data from another system it has some strong weaknesses. The most important one is latency; which can become high (conflicts with requirement ReD-IM2). As a result the data in the visualization is only as up to date as the latest pull interval. One solution would be to reduce the pull intervals to a very small value like every second. However, pulling data from a system typically generates a high load in the system. Therefore, shortening the intervals will lead to performance degeneration in the data provider which would violate requirement ReD-IM3.

Another weakness of this solution is the increased effort to implement the data adapter. Nonetheless, it is somewhat compensated by the fact that no plugin needs to

[7] In the context of BI systems these scheduled jobs are referred to as ETLs for Extract, Transform, and Load which are the three main steps of the jobs

be implemented for it. Because of these deficiencies Pull-Forward data adapter in an EMI should be reduced to a minimum. If no plugin mechanism is provided by the data provider, however, a Pull-Forward data adapter is sometimes the only possible choice. Yet, sometimes another system related with the data provider offers a plugin or web-hook mechanism. In this case an Invoke-Pull data adapter is the better alternative.

Invoke-Pull

Data providers are typically connected with each other. For example a good practice in software development is to tag a commit into a version control system (VCS) with the task number of a task in a change request management system (CRM). The number of changed files per task could be used as a complexity measure for the task. Additionally, the number of changed lines of code could be used to normalize the effort for a task. Of course, every commit alters the number of changed files for a task. Hence, after every commit a special data adapter needs to send a new message to the EMDB containing additional information to the task. This then allows a special metric kernel to calculate the two measures.

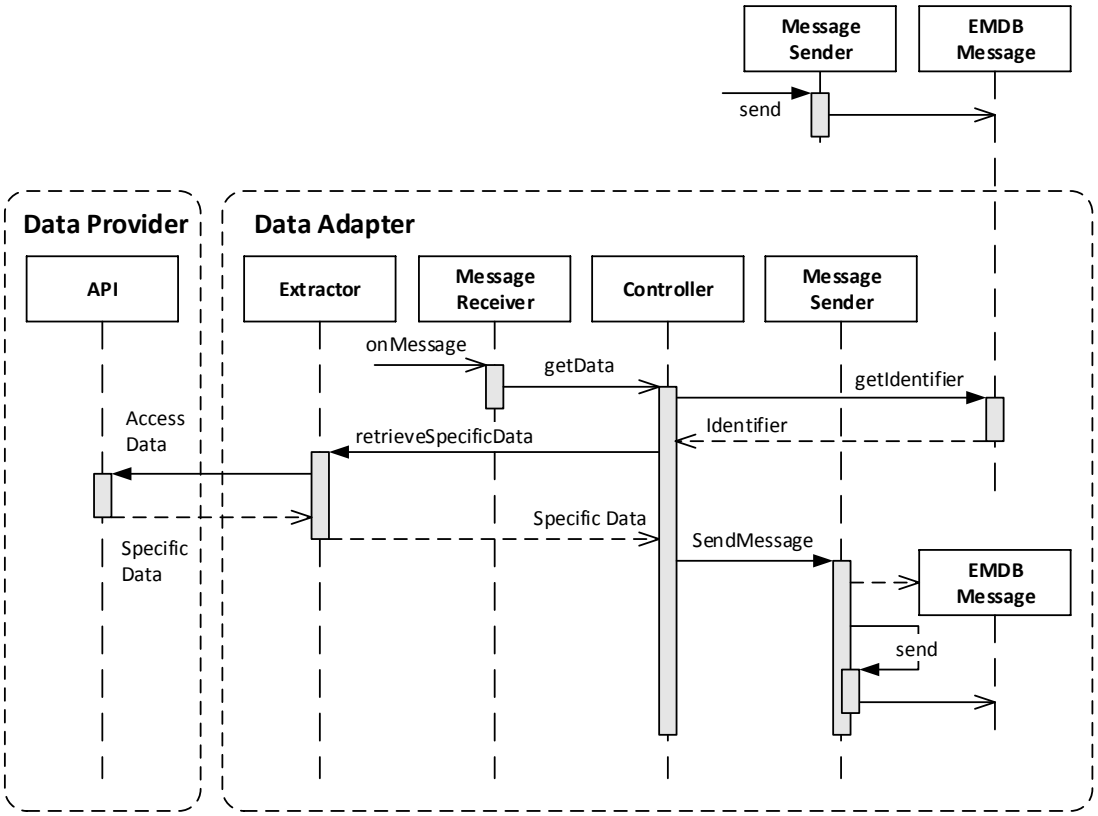


Figure 5.25.: UML sequence diagram for the concept of the invoke-pull adapter pattern

Additionally, many data providers nowadays provide a so called *web-hook* mechanism,

which can be fired on certain events in the data provider like *on data change*. The data provider will call the URL in the web-hook whenever the configured event occurs. Unlike plugins, these web-hooks typically only allow to transport a data identifier for the changed data and not its complete data. Therefore, another part of the EMI needs to retrieve the data from the data provider when the web-hook triggers it. However, we like to decouple these two functions because the actual change event may be used for other purposes as well and not only the data retrieve trigger.

Figure 5.25 shows the interaction of the different parts of the invoke-pull adapter pattern as UML sequence diagram. This pattern enables EMI developers to implement a data adapter that utilizes an event as a trigger. As described above, the data provider will trigger an event gateway in the EMI which generates an event on the event topic of the EMDB. This event is then received by a data adapter which extracts (pulls) the changed data from the data provider. This data is then included in an EMDB Message and send to the EMDB. Because this data adapter pattern does not rely on a timer but uses an event as trigger its latency is also very good (requirement ReD-IM2). It is, however, slower than a push-forward adapter because of the additional event that needs to be send and received. Like all other data adapter this one also only uses APIs of the data provider. It is, hence, also decoupled from the data provider fulfilling requirement ReD-IM3.

Invoke-Dump

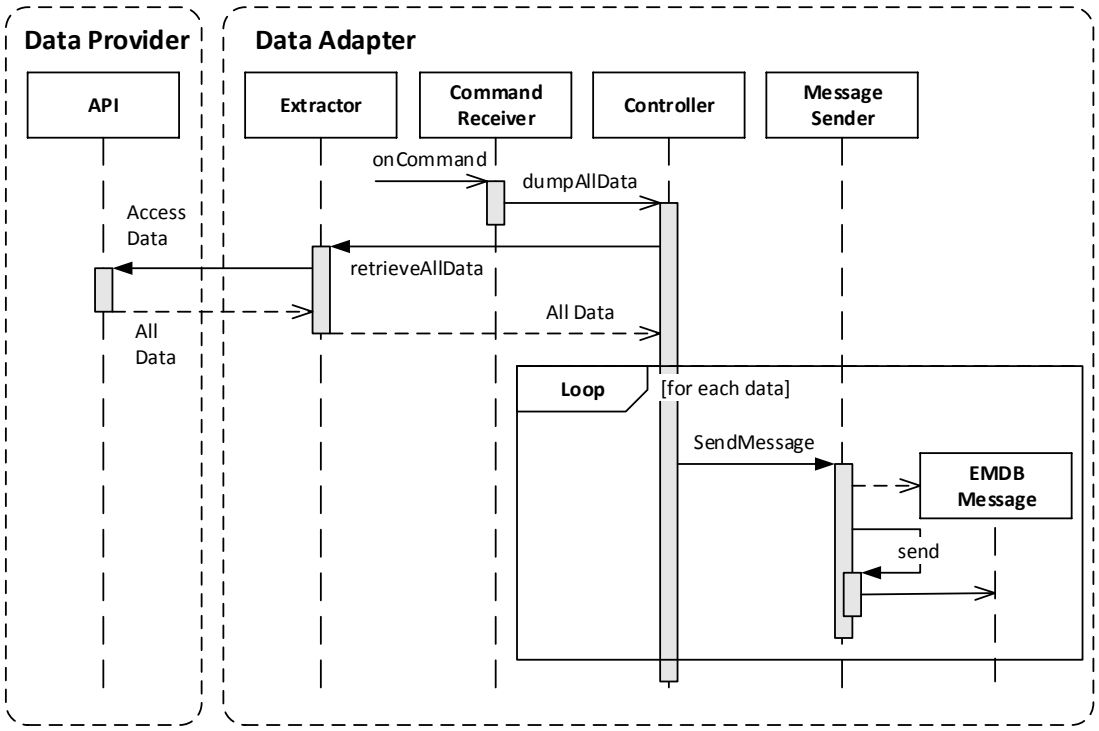


Figure 5.26.: UML sequence diagram for the concept of the invoke-dump adapter pattern

Setting up an EMI or building a dedicated data analysis tool based on our reference architecture often requires to get all data from a data provider with one action. This one action will then *dump* all the data from the data provider to the EMDB. Figure 5.26 shows the interaction of the different parts of the invoke-dump adapter pattern as UML sequence diagram. The idea behind the pattern is similar to an invoke-pull data adapter. The trigger for the pull, however, is a dedicated command message that is transported over the command-topic of the EMDB. Dumping all data will create a high load on the data provider. Therefore, it is important to ensure that the dump is not triggered accidentally but only when definitely required. Upon receiving the command the data adapter will access the APIs of the data provider to get all relevant data from the data provider. It will then send multiple EMDB messages to the EMDB. One message for each dedicated chunk of data. Hence, the metric kernel do not need to implement additional methods to handle complete dumps because the multiple messages look like all the other data from the other data adapters.

This data adapter influences the data provider due to heavy load because of the dump which violates requirement ReD-IM3. Also the data is far from real time; violating requirement ReD-IM2. The invoke-dump pattern, however, is only used for special situations very sparsely.

After these dynamic views on all the data adapter patterns the next section provides static reference architectures for the internal components of the different types of data adapters.

5.4.2. Static Reference Architecture

The dynamic views in the last section already provide an inside into the internal components of the different types of data adapters. This section will elaborate on this and provide two static reference architectures for the different types of data adapters. The first subsection provides the static reference architecture for Data Gateways from the push-forward data adaption pattern. The second subsection provides the static reference architecture for the pull-based data adapters.

Data Gateway

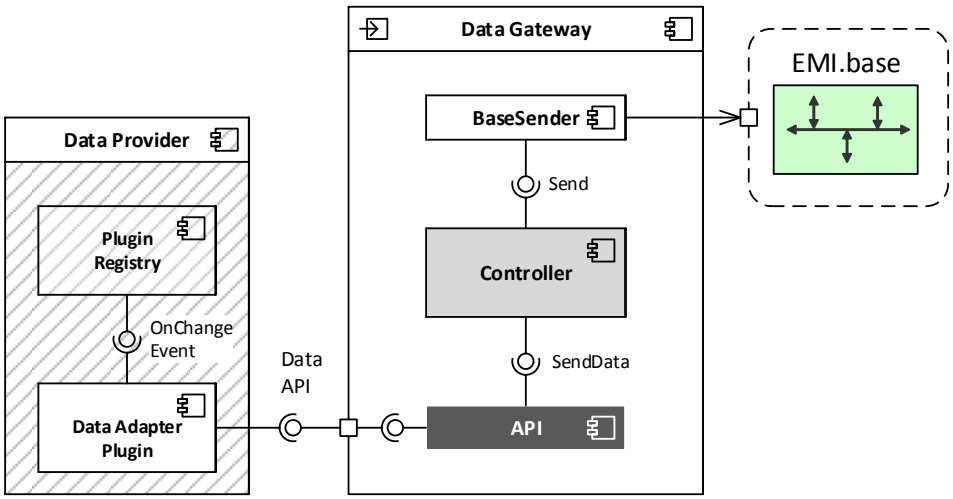


Figure 5.27.: Static reference architecture for Data Gateways

The Data Gateway static reference architecture applies for the push-forward adapter pattern. The sequence diagram from above already indicates some of the core inner components for the data gateway. However, figure 5.27 provides a complete overview over all the recommended inner components of the data gateway and the plugin in the data provider. The push-forward data adapter pattern utilizes the idea of inversion-of-control in which the data is pushed to the EMI from the data provider (the active component is the data provider and not the EMI). The data adaption therefore starts in the data provider itself. The EMI requires a specific plugin in the data provider that hooks into the plugin mechanism of it. This plugin registers itself to be called on data change (including creation and deletion) similar to the well known observer pattern [GHJV95]. The plugin then calls the `Data API` on the Data Gateway which delegates the call to the `Controller`. The controller transforms the data so it can be send to the EMDB using a message sender. We included the base sender in the figure because most of the data gateways send data to the base topic of the EMDB.

Pull-based Data Adapter

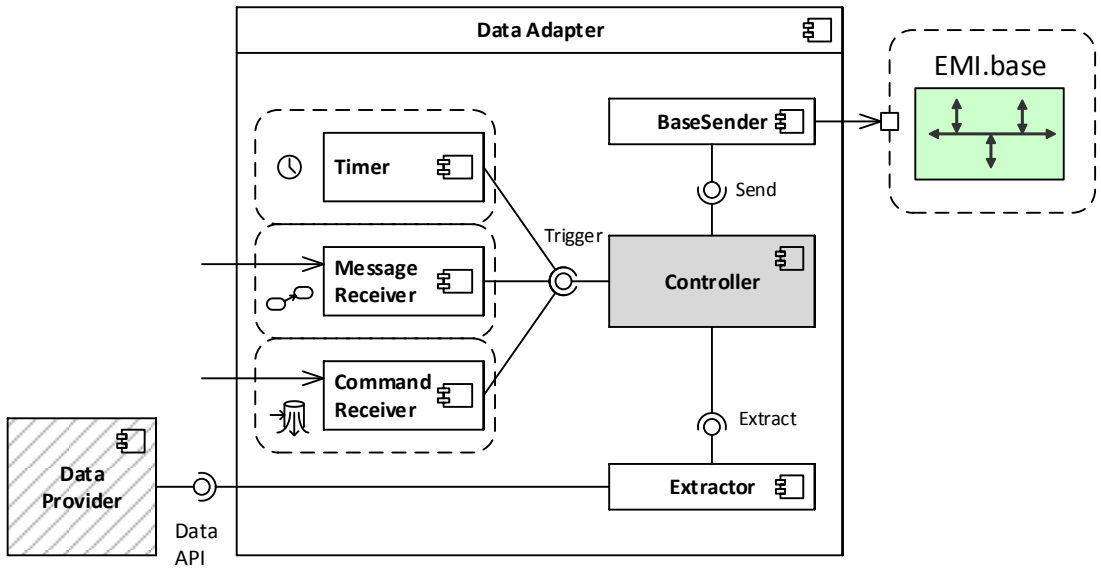


Figure 5.28.: Static reference architecture for pull-based data adapter

All other types of data adapter share a similar static design. They only differ in one small component: the trigger. Figure 5.28 provides an overview over all the inner components of the pull-based data adapters. As indicated by the sequence diagrams from above. They all require an extraction component (`Extractor`) that gets the data from the data provider. Therefore, the data provider needs to grant access to the data for example via a REST API, SOAP Web Service, or, more dirty, its database. The extract component provides the central `Controller` component of the data adapter with all the functions required to get the data (single or multiple data entries). Similar to the Data Gateway above the controller then transforms the data and hands it to the Message Sender for sending to the EMDB. Contrasting the Data Gateway from above, the controller provides a trigger interface that can be used by the various triggers required to implement the various adapter pattern.

The Pull-Forward adapter pattern utilizes a time component to trigger the extraction of the changed data as described in the sequence diagram above. The Invoke-Pull adapter pattern requires a message receiver that listens on the EMDB (most of the time it will connect to the event topic). Following the sequence diagram from above the message receiver will then extract the data identifier from the message (event) and specifically triggers the controller. The last adapter pattern invoke-dump also requires a message receiver for the EMDB. Contrasting an invoke-pull adapter, however, the invoke-dump pattern requires a command receiver. The receiver will trigger to full data extraction of the data provider in the controller upon receiving a valid command message. The command message can also include additional configuration information for example to only trigger partial dumps or specific dumps. Event though figure 5.28 shows all the

different trigger in one data adapter. In an EMI instance typically only one of the trigger options is implemented.

5.4.3. Summary

This section introduced different adapter patterns to implement data adapter in an enterprise measurement infrastructure. We first introduced the different integration problems that need to be faces by the data adapters. After this, we then introduced the dynamic behavior of the different adapter pattern. After that we also provided two reference architectures for the internal components of the data adapters. Similar to the services in the previous sections these reference architectures provide blueprints for the actual architecture of the data adapters in an EMI instance. Unlike the services in the previous sections, however, multiple data adapters can be instantiated for the adaption of different data providers. Also the pattern only provide guidance for the design. An actual data adapter can implement multiple patterns if this is required in the actual EMI instance. The next section will introduce the reference architecture for the *heart* of the EMI: the metric kernels.

5.5. Metric Kernel Reference Architecture

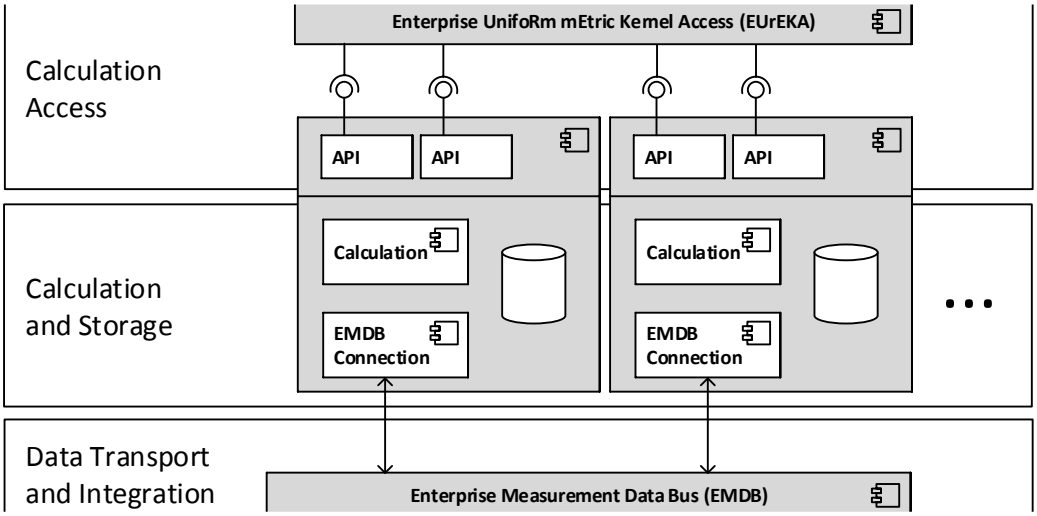


Figure 5.29.: Zoom into the metric kernel related layers of the MeDIC reference architecture

Metric kernels provide the core functionality of the measurement infrastructure: the actual calculation of metrics. Furthermore, the metric kernels also implement all the associated concepts of (derived) metrics like the variability model and the measurement producers. Hence, metric kernels are the most important part in an EMI. Figure 5.29 shows an overview over the calculation and storage layer of the reference architecture. From a logical point of view the indicator access APIs of the metric kernel belong to the calculation access layer even though they are provided by the metric kernel and often physically deployed with them in one component. The core of the metric kernels are the metric calculation components, the data storage, and the connection to the EMDB in the bottom.

Our discussion about microservices in the introduction of this part focused on the benefits of being able to choose technologies for each microservice independent from each other. Metric kernels benefit hugely from being microservices and this fact.

In this section we will first introduce the static reference architecture for metric kernels in the following subsection 5.5.1. In this we present two different options for the layout of a metric kernel. After that we will introduce the dynamic aspects of the metric calculation and the indicator access in more details in subsection 5.5.2. As usual we close the section with a summary in subsection 5.5.3.

5.5.1. Design Alternatives

This subsection presents two different alternatives for the reference architecture for metric kernels. Both alternatives provide similar dynamic behavior and use similar components. However, the first one is a monolithic design with just one high level component for the whole metric kernel. The second one is a design with two separate components for the two main tasks of the metric kernel: data reception & pre-processing and metric calculation. Hence, these differences are mostly visible in the static view. Most of the time we found the monolithic design sufficient and the kernel was not too large to maintain. However, we also used the other design in some of our field studies when the kernels provided a lot of different metrics. We start the description with the monolithic design in the following sub section.

Monolithic Metric Kernel

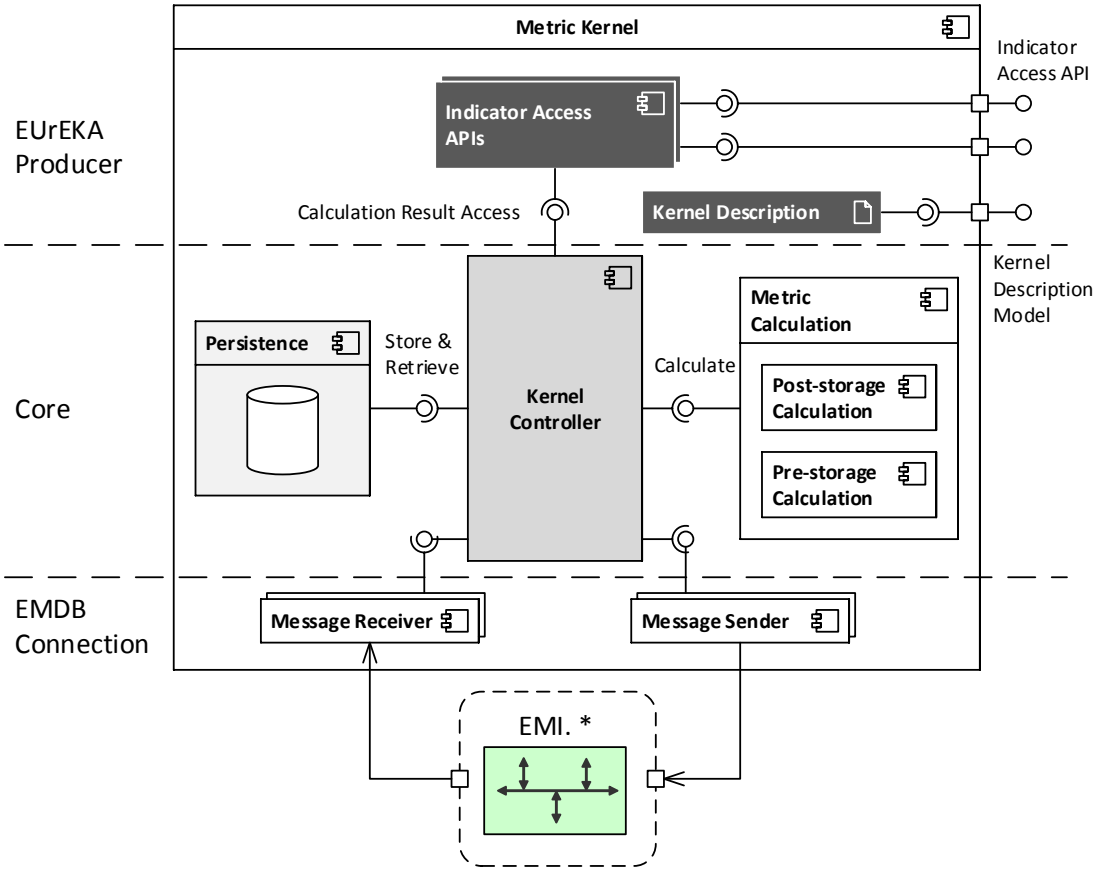


Figure 5.30.: Static monolithic metric kernel reference architecture.

Figure 5.30 shows a detailed view on the recommended internal components of our monolithic metric kernel design. The design features three layers that reflect the

surrounding layers in the MeDIC reference architecture. The metric kernel connects to the EMDB in the bottom layer using `Message Receiver` and `Message Sender` components. As discussed with the message cache in section 5.2.4 these components are most likely offered by the development platform of the EMI instance.

The central core layer implements the storage and calculation aspect. The central `Kernel Controller` component orchestrates all calls from the EMDB and the indicator access API. The controller also triggers the different calculation tasks and communicates with the `persistence` component. The `Metric Calculation` component realizes the actual calculation of the metrics.

The calculation component has two parts: a pre-storage calculation and a post-storage calculation. The idea is that it is often wise to transform the incoming data from the EMDB into a different format and perform additional calculations on them before they are persisted. This can speed up and ease the calculation of the actual metric values upon request by the indicator access API significantly.

The `Indicator Access APIs` and the `Kernel Description` are located in the EUrEKA producer layer of the metric kernel. Section 5.3 provides additional details on the design of the APIs and the kernel description meta model. The post-storage calculation component is also utilized to calculate the data that is produced following the measurement producer specification (see section 2.3.7 for more details) of the metric kernel and send to the EMDB.

Separated Components

Figure 5.31 provides an alternative design for the static component architecture of the metric kernel. This design splits the two logical parts of the metric kernel into the `Metric Kernel Receiver` component and the `Metric Kernel Access` component. The metric kernel receiver component houses the `Message Receivers` to connect the kernel to the EMDB as well as the pre-storage calculation component. The `Kernel Receiver Controller` orchestrates the control flow and triggers the calculation and storage in an external storage component (`Persistence`). After the pre-storage calculations are performed and the data is stored the controller will call the `Production Trigger` on the metric kernel access component. This will trigger the production and sending of all measurements defined in the measurement producers (see section 2.3.7 for more details). Furthermore, the metric kernel access component houses the EUrEKA producer layer similar to the monolithic design from above as well as the post-calculation component and the `Kernel Access Controller` for orchestration.

A benefit of the separate components design for metric kernels is that the technologies to implement the two components not necessarily need to be the same. Although this may require to extract the message senders into their own component if the messaging technology is hard to access from the technology chosen for the metric kernel access component (e.g. JMS from Javascript).

A weakness of the design, however, is the integration between the two components via the storage component in the middle. This strongly couples the two components and a change in one of the components that require persistence changes will inevitably also

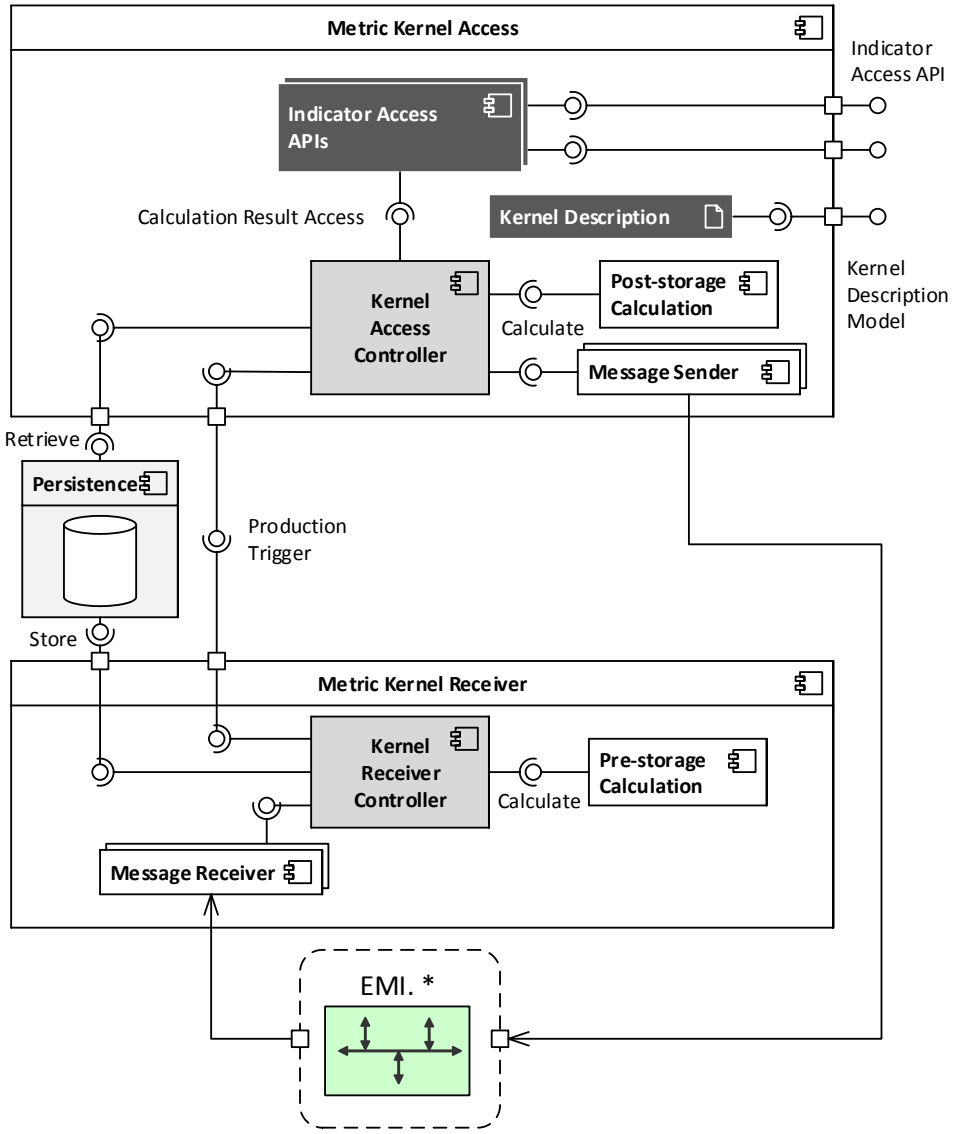


Figure 5.31.: Static metric kernel reference architecture with separated components.

require changes in the other component. This is a potential source of errors which need to be addressed in the actual maintenance procedures of the EMI instance that utilizes this design for one of its metric kernels.

When instantiating our reference architecture to an EMI instance it is important to choose the specific metric kernel design (monolith or separated) for each metric kernel separately. Also note that a monolithic metric kernel can be refactored to a separated metric kernel if the kernel becomes to large. Alternatively, sometimes a metric kernel needs to be separated into two individual kernels if it is required to perform diverse calculations.

This concludes the rough description of the two different design alternatives for the metric kernels. The next section will provide additional details on the dynamic processing of measurement messages and metric calculation inside a metric kernel.

5.5.2. Dynamic View

The previous section already discussed some dynamic aspects of the metric kernel when presenting their static reference architectures. This section will provide additional details to the behavior of the metric kernel. The first subsection will present the message processing behavior. This presents the activities that are executed when a metric kernel receives data from the EMDB. The second subsection provides additional details to the calculation of the metrics on request by one of the indicator access APIs.

Message Processing

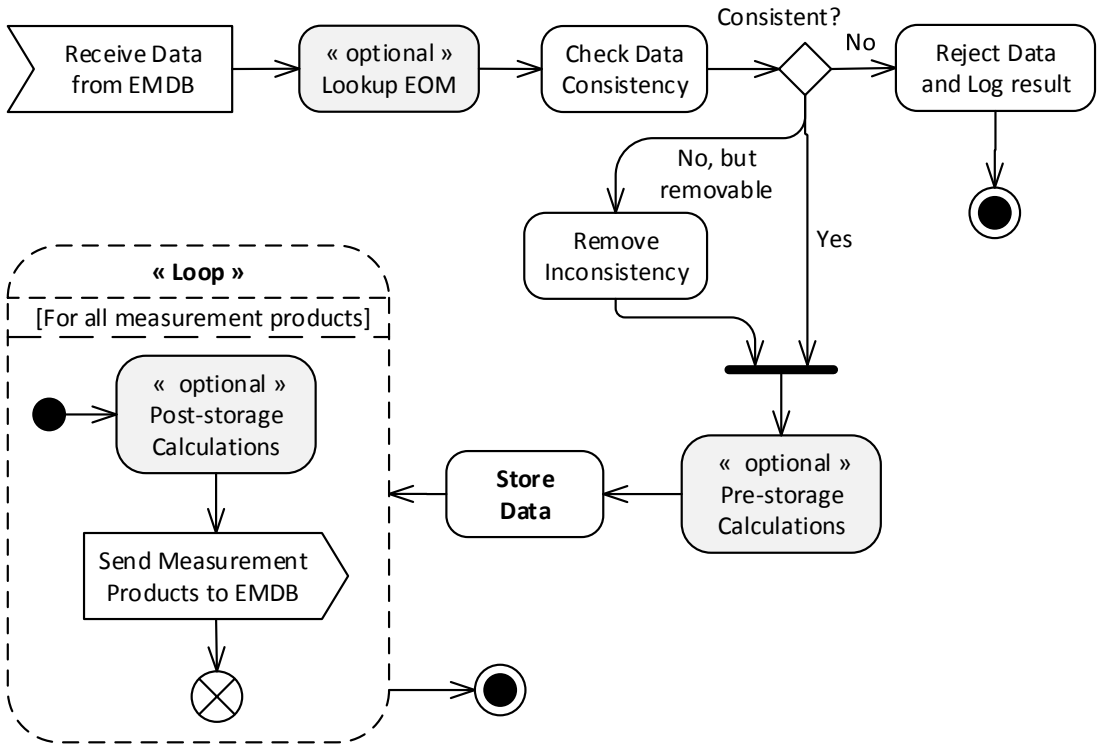


Figure 5.32.: EMDB Message Processing

Figure 5.32 shows a UML activity diagram for the prototypical activities in the message processing of a metric kernel. Some of the activities are marked by the «`optional`» stereotype. These activities, as the name suggests, are optional for the processing of EMDB messages in the metric kernel. Additional steps can be inserted everywhere in the flow when instantiating the reference architecture in an actual EMI if it is required.

The flow starts with the data reception in the metric kernel. The data is received by an EMDB message receiver. This receiver can either receive pure measurement data, base data, events, or other data on specific topics of the particular EMI. After the data is received the metric kernel can use the directory service to translate synonyms; for example the identifier for the entity of measurement or status identifiers. Due to our operation activities related to failures in an EMI described in section 12.2.2 it is important that the metric kernels resolve the synonyms and not the data adapters (which would then send the measurement message with the specified term). Another alternative would be to resolve the synonym upon request via a indicator access API. This would ensure that a synonym is always correctly resolved event after updating synonym-term relations without any additional actions. However, this would slow down the answers to the requests of the indicator access APIs when resolving the synonym requires communication via the messaging system. Nevertheless, if these are cached as proposed in the reference architecture for the lookup system (see section 6.3) then the performance loss is neglectable and synonym resolving should be performed in the indicator access API and not in message processing if possible[8].

Most importantly the kernel then needs to check the data consistency before processing and storing the data. We differentiate between three different outcomes for the consistency check:

Inconsistency not removable If an inconsistency is detected which is not removable then the metric kernel needs to aboard the data processing and reject the data. This needs to be logged; including the inconsistency findings and the message details because it may require additional actions from the operator or other stakeholders. Furthermore, this may put the kernel in out-of-sync state if the data in the kernel persistence does not reflect all data in the data provider.

Inconsistency removable Removable inconsistencies need to be removed before continuing with the data processing and storage. Missing data that can be restored from the metric kernel storage is an example for a removable inconsistency. This still needs to be logged to indicate potential data-quality problems because removal is only based on heuristics!

No inconsistency If the data is consistent the metric kernel can proceed the processing and storage.

After the consistency check the metric kernel may perform optional pre-storage calculation actions. These actions can include data transformations or actual (pre-) calculations on the data before storage. The aim of this step is to store the data in a way that the calculation of the metrics based on this data is as simple and efficient as possible when the metrics are accessed from the indicator access APIs. The data is stored in the persistence component of the metric kernel.

[8] Resolving the synonyms at this point may require to change database queries and calculation logic because it does not only need to consider a single eom but a collection of eom synonyms. This may be impractical and complex and resolving the synonyms should be performed on message reception.

After this the metric kernel needs to produce all the measurements required in the measurement producers for the metrics of the metric kernel. For each of these producers the metric kernel may perform additional calculation actions before sending the calculation results (the measurement products - see section 2.3.7) to the EMDB.

Indicator Access

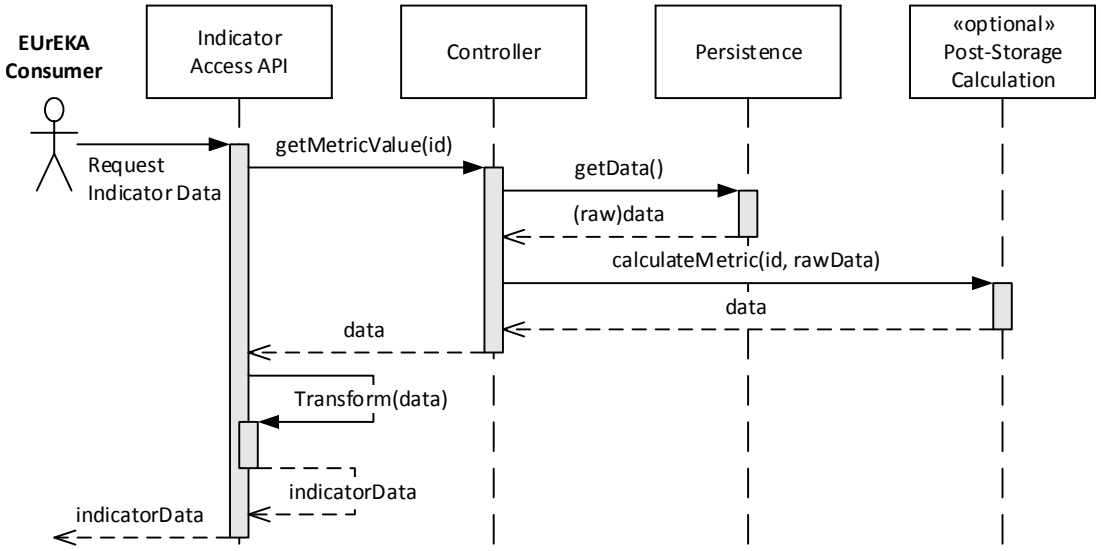


Figure 5.33.: UML sequence diagram for the behavior of the metric kernel components when data is requested via an indicator access API

Figure 5.33 provides an UML sequence diagram for the prototypical activities performed when an EUrEKA consumer accesses an indicator access API of the metric kernel. A EUrEKA consumer requests the calculation results via a specific metric data API of the metric kernel. The API then delegates the call to the controller which retrieves the necessary data from the data storage. Optionally, the retrieved data is then feed to the post-storage calculations component. After the calculation the controller returns the result to the indicator access API component. The API then transforms the data into the required data type. This may require to wrap it or transform it into new data transport objects. The indicators are then returned to the EUrEKA consumer which visualizes them.

5.5.3. Summary

This section presented the reference architecture for metric kernel. This included static and dynamic aspects. For the static aspect we presented two different designs for the component composition inside the metric kernel. The monolithic design is suitable for typical small to medium sized metric kernels that only calculate a small number of metrics.

The separated design is suitable for larger metric kernels. The separated design follows the microservice more closely with dedicated components for the two main tasks of the metric kernels: data reception & storage and indicator calculation. However, the design also increases the structural complexity of the metric kernel and raises the maintenance effort due to close coupling of the two components via the persistence component.

After this we presented prototypical behavior fragments of metric kernels. Thereby we investigated the most crucial activity when receiving EMDB messages in a metric kernel: the consistency check. We differentiated between not removable and removable inconsistencies and provided examples for these. We also provided further insides into the production of measurements based on the measurement producer definition from the metrics which the kernel implements.

The following section will introduce our reference architecture for the consumers of the indicators provided by the metric kernels: the frontend components in the visualization layer of the EMI.

5.6. Visualization Reference Architecture

The previous section introduced the reference architecture for metric kernels; the implementation of the metrics in an EMI. Following our formalism and terminology in section 2.1.1 and section 2.3 respectively, the metrics produce indicators which are consumed by visualizations. This section will provide the reference architecture for the visualization frontends that provide the monitors that implement the visualizations (see section 5.1.2).

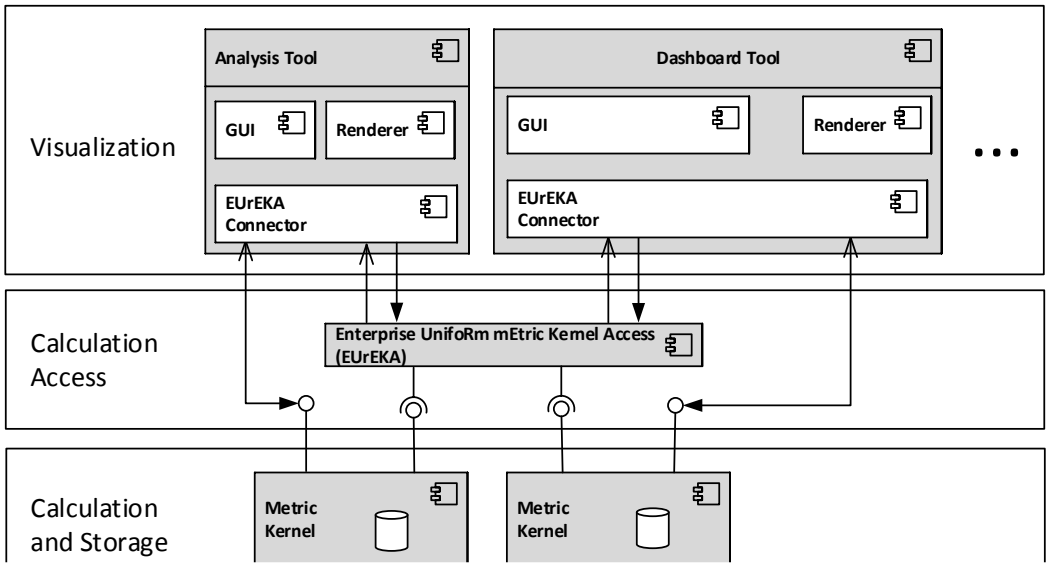


Figure 5.34.: Zoom into the visualization layer of the MeDIC reference architecture

Figure 5.34 provides an overview over the core components of the visualization frontends in the visualization layer as well as the related layers underneath. The most important components in the visualization frontends are the graphical user interfaces (GUIs). These utilize the renderers to visualize the monitors that answer the information needs of the metric customers. The data for the monitors is provided by the EUrEKA consumer component. This connects to the indicator access APIs as well as the EUrEKA registry service (see section 5.18).

Requirement ReD-IV3 requires specific support for dashboards and analysis tools. The following subsection 5.6.1, therefore, first discusses the term *dashboard* and provides the basis for our term *metric-based monitoring dashboard*. Following this, subsection 5.6.2 provides further details on the classification of the visualization frontends into M^2 dashboards and analysis tools. After the classification we provide the static reference architecture for visualization frontends in subsection 5.6.3 and provider further insides into the dynamic aspects of the monitor creation in subsection 5.6.4. As usual we close the section with a short summary in subsection 5.6.5.

5.6.1. Metric-based Monitoring Dashboards

Today the terms dashboard is used inflationary throughout several applications. According to Few a dashboard is "a visual display of the most important information needed to achieve one or more objectives; consolidated and arranged on a single screen so the information can be monitored at a glance." [Few06]. This definition shows the importance of the monitoring aspect within dashboards. Analogous, Fitch defines dashboards as "a way to collect, summarize, display and manage by a highly tuned set of business performance metrics across a complex enterprise." [Fit]. This definition shows the importance to collect the correct metrics because they are the basis for decisions and management. The definitions also aligns well with the general ideas of performance measurement systems [The97]. Similar to and sometimes synonymously used with dashboards: balanced scorecards also provide a monitoring concept for an organization [KN92]. However, according to Horvath and Kaufmann they focus more on the implementation of business strategies and influences between the metrics [Eck06]. Additionally, cockpits and software project control centers include "support for systematically deriving the right control mechanisms for a project and organization based on context information and organizational goals" [HM08a].

Shelby noted an overwhelming amount of dashboard definitions and usages [Sel05]. Therefore, in his work he called them measurement-driven dashboards. This includes the metric aspect and also underlines measurements as a basis and driving factor behind the development of the dashboards. However, we like to root the dashboard on the information needs (the monitoring-needs) of the stakeholders. Thus, we chose the prefix metric-based monitoring in contrast to measurement-driven. This emphasizes the monitoring aspect from Few as well as our metric emphasize as driver for the monitors like supported by Fitch. Therefore, to avoid confusion, we call the dashboards addressed in this thesis: metric-base monitoring dashboards (or M^2 dashboard for short).

5.6.2. Visualization Frontend Classification

Stephen Few classifies different types of dashboards in his book Information Dashboard Design [Few06]. He differentiates between *analytical*, *strategic*, and *operational* dashboards. The difference between these different types of dashboard is the focus of the information needs that is answered in the dashboards. As we described above (requirement ReD-IV3), we like to classify the visualization frontends on a more coarse grained scale. We like to differentiate between *metric-based monitoring dashboard (M^2 Dashboard)* and *Analysis Tools*.

Metric-based monitoring dashboard The goal of M^2 dashboards is to provide dashboards that allow metric customers to quickly get an overview over many information needs. Mostly these information needs focus on just on entity of measurement. For example a dashboard to get an overview over a specific project of the company. The M^2 dashboards are accessed by multiple metric customers. Hence, they often use visualizations that are easy to understand by the different

metric customers. A M^2 dashboard tool should support the management of the dashboards including a way to easily configure dashboards based on information needs.

Analysis Tool The goal of analysis tools is to get deeper insights into a specific detail. Hence, they only answer a very narrow and limited set of information needs. The analysis tools are typically only used by specially trained metric customers. Thus, they often use very specific and specialized visualizations to allow experts better and quicker analyses. The analysis tools do not need dedicated dashboard management because they typically show a fixed set of visualizations. They do, however, need a means to provide control over certain aspects of the data feeding the visualizations like filters and selection mechanisms to enable exploration.

Even though the two types of visualization frontends differ quiet a lot they also have a lot of aspects in common. The core of both tools is satisfying information needs of metric customers based on monitors that visualize indicators from metrics. Both tools also need to be flexible and extendable to reflect changes in information needs or new needs (requirement ReD-IV2). From the technical view of the EMI both types of visualization frontends need to consume indicator data from the calculation access layer (requirement ReD-IV1). Therefore, the following reference architecture is applicable to both types of visualization frontends.

5.6.3. Component View

Figure 5.35 provides an overview over the recommended internal components of a visualization frontend. The metric customer interacts with the `Graphical User Interface` on the top. This visualizes the dashboards and the analysis frontend. They both contain monitors that visualize indicator data. The GUI utilizes several `Renderer` to create the monitors. Renderer visualize certain data typically using dedicated rendering libraries like Data Driven Documents (`http://www.d3js.org`) or JFreeChart (`http://www.jfree.org/jfreechart`). Furthermore, the GUI can check certain access rights of users via the optional `Authorization and Authentication` (A&A) component. Besides from the direct connection to the renderer and the utilization of the A&A component the GUI delegates all calls to the `Visualization Controller` which orchestrates the component interaction. In a M^2 dashboard the controller delegates the dashboard management tasks (create, update, delete, share, clone) to the optional `Dashboard Management` component. The controller uses the `EUrEKA Connector` for the integration with the indicator access APIs and utilization of the EUrEKA registry. Furthermore, the controller and the dashboard management component may use the A&A component to check authorization of certain activities.

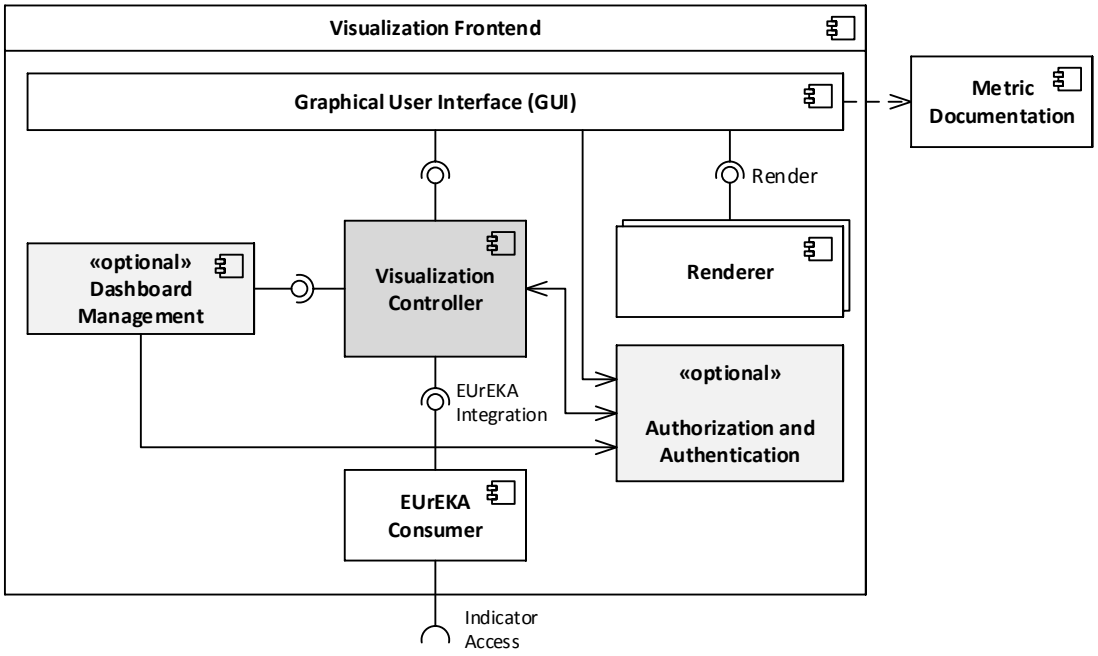


Figure 5.35.: Dashboard Application - Component View

An analysis tool will most likely not implement the optional components (dashboard management and A&A). It requires a more complex interaction in the GUI to filter and select the data for the visualization as well as navigation between different views on the same data. The filtering and selection will typically use the variability model of the metrics as a source for the different filters. The visualization frontend can then include specific variability configurations when the data is requested to apply the filters.

Another important outside connection for the visualization frontends is the connection to the metric documentation system. This enables the GUI to show interpretation aids right next to the monitor. The documentation can also act as a source for monitor definitions if the source of the documentation is a formal metric model that includes the specification of monitors.

5.6.4. Dynamic View

The previous section already discussed some dynamic aspects of the visualization frontends when presenting their static reference architectures. This section will provide additional details to the behavior of the M^2 dashboards and analysis tools. We already described the differences between the two types of visualization frontends. Regarding the static reference architecture the M^2 dashboards will most likely use the optional components whereas the analysis tools will not. The analysis tools, however, require a more complex interaction regarding the variability model and variability configuration of the metrics. M^2 dashboards may also offer the definition of variability configurations for certain

monitors. However, the mechanism is typically used less than in analysis tools. Thus, we need to focus on different aspects when defining the behavior of the two different types of visualization frontends. Therefore, in this subsection we first discuss the behavior of the M^2 dashboards including the optional components and then focus on the analysis tools which use variability configurations.

M^2 Dashboard Dynamics

Figure 5.36 presents the prototypical behavior of a M^2 dashboard when preparing monitors as UML sequence diagram. The sequence starts with the metric customer who wants to satisfy some of her information needs. Hence, she requests one of her M^2 dashboards. First the visualization controller needs to lookup the user and check if she is authorized to look at the requested dashboard. If the access is not granted the controller generates an error message and returns this back to the visualization. This security steps are optional and only required for organizations that want to restrict the access to certain data. After this the M^2 dashboard configuration needs to be retrieved from the dashboard management component if it is not a fixed M^2 dashboard. The controller then needs to retrieve the indicators for each monitor on the M^2 dashboard. The data retrieval is delegated to the EUrEKA consumer component. This returns the data back to the controller which then returns all the indicator data back to the GUI. The GUI then utilizes the renderer to render all the monitors on the M^2 dashboard and then presents it to the metric customer.

Analysis Tool Dynamics

Figure 5.37 presents the prototypical behavior of an analysis tool when preparing monitors as UML sequence diagram. Similar to a M^2 dashboard the visualization in an analysis tool is triggered by an information need of a metric customer. Contrasting the need from above, however, the needs are analysis focused and typically vague. For example “Is there something wrong with our change request management process and if yes: what is wrong?”. To satisfy these needs the metric customer needs to look at multiple data from different perspectives. Therefore, she needs to change the variability configurations of the indicators feeding the monitors. The first action of the controller is therefore to check the validity of the variability configuration argument. It delegates the check to the EUrEKA connector which asks for an instance of the particular variability model and delegates the check of the configuration to the model. If the configuration is valid the sequence continuous similar to the sequence above. The controller requests the indicator data (with the specific variability configuration) and the GUI then utilizes the renderer to create the monitors in the view.

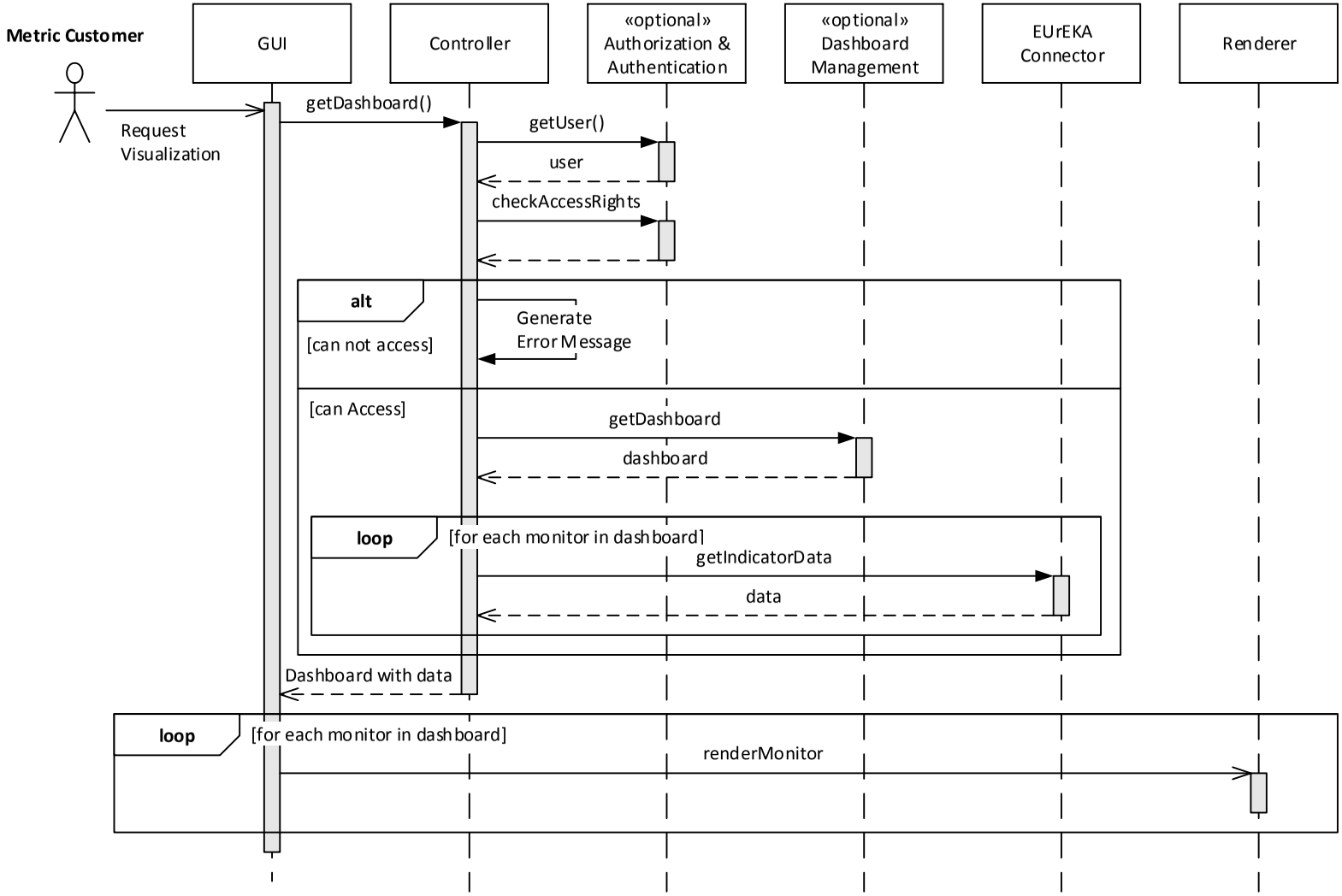


Figure 5.36.: Typical M^2 dashboard visualization dynamic as UML sequence diagram

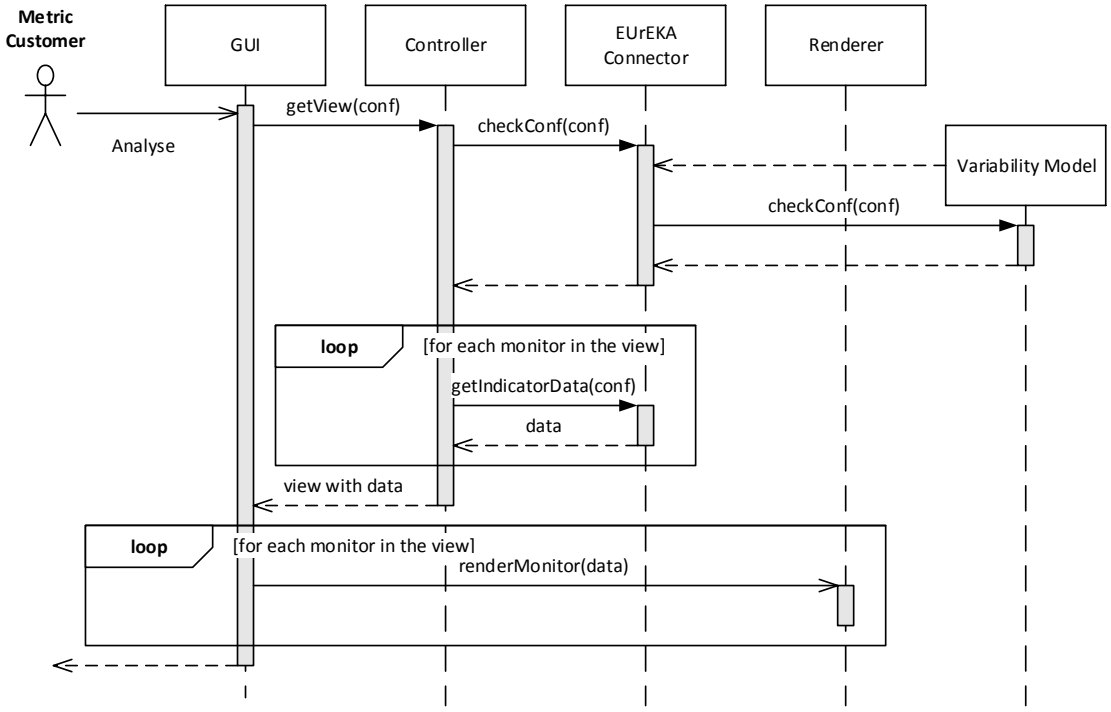


Figure 5.37.: Typical analysis tool visualization dynamic as UML sequence diagram

5.6.5. Summary

This section defined our reference architecture for visualization frontends. We started the section with a discussion on the arbitrary use of the term dashboard. We also motivated our term *Metric-based Monitoring Dashboard* (M^2 Dashboard) for the dashboards considered in this thesis. Our classification also described the idea behind the analysis tools. Contrasting the M^2 dashboards, the focus of an analysis tool is to gather additional insides into existing information needs or investigating an entity of measurement looking for new specific information needs (and optimization potential or problem sources).

From there we started with the definition of the static component-based reference architecture for visualization frontends. In the reference architecture we did not differentiate between the different types of dashboards. We rather placed optional components in the reference architecture that are more likely be used by M^2 dashboards then analysis tools. However, M^2 dashboards do not need to use them and analysis tools may want to use some of them; for example the A&A component to restrict the access to certain data. The reference architecture uses renderers to visualize the monitors and utilizes a EUrEKA connector component to access the indicator data for the monitors.

Following the static reference architecture we presented two behavior sequences. One focused more on the interaction in M^2 dashboards including dashboard management and A&A whereas the second one included variability configurations which are more likely to be used in an analysis tool.

5.7. Technical Integration of Operation Services

This section introduces our prototypical solution for the integration of operation services and EMI services. The ideas presented here are used in the definition of the actual reference architectures for the monitoring system (section 6.1), the logging system (section 6.2), and the directory system (section 6.3). Hence, we refer to this solution as *style* rather then reference architecture. It will focus on the static architecture and the integration aspects. The concrete reference architecture of the operation services will then further specify all the components and their details as well as the behavior of the components.

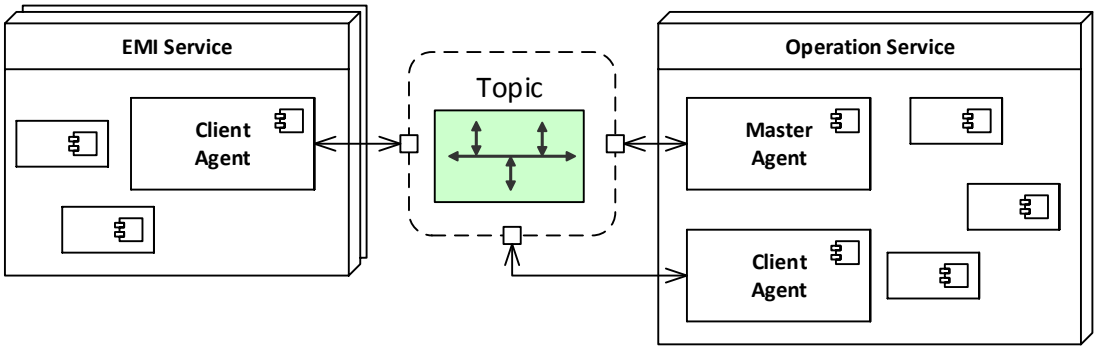


Figure 5.38.: Integration in an Operation System between the Operation Service and EMI Services

We discussed different integration architectures in section 3.3. In our technical reference architecture we choose to integrate the different layers using our enterprise measurement data bus for the integration between the measurement and calculation layer and the EUrEKA design for the integration between visualization and calculation. These integration architectures nicely fulfill all our requirements. The integration between the operation services and the EMI services, however, requires a different type of integration architecture because of different control and data flows between the services. Additionally, EMI services should be easily integrated with the operation services using common modules and components provided by the specific EMI development platform.

We therefore choose an agent-based integration architecture for the integration between EMI services and operation services. Figure 5.38 provides an overview over the core components for this design. Each operation service defines a specific topic to integrate the agents with each other. Each operation service consumer utilizes a specific *Client Agent* to connect to the topic and integrate with the operation service. The integration service provides a *Master Agent* which orchestrates all other agents and provides the integration for the operation service. The communication between the agents and the topic is bi-directional contrasting the EMDB and the EUrEKA design which are both uni-directional. This enables a uniform communication between all types of agents. Yet, it increases the complexity of the actual agents and requires additional infrastructure services (the additional topic). However, a thoughtful design of agent-frameworks provided by the platform eases their design and decreases the complexity from our experience.

The additional topics are also easy to handle because the EMDB already uses at least three topics for the core communication inside the EMI.

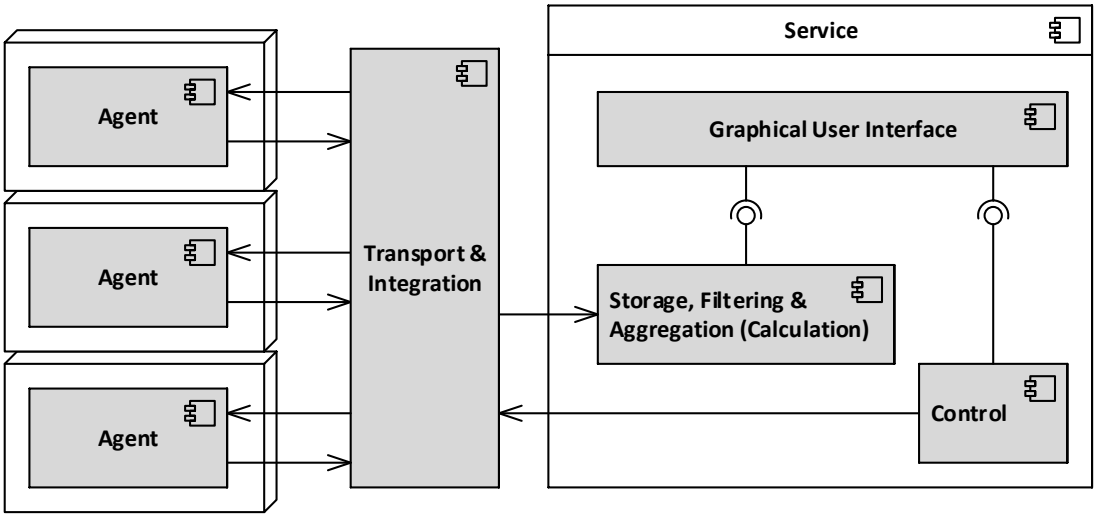


Figure 5.39.: Operation System Component View

Figure 5.39 provides additional details to the description from above. Each operation service provides specific agent components that are deployed with each integrated service. The actual integration of the agents is performed by the transport and integration component which utilizes a specific topic for each operation service. Furthermore, this component provides the agents with the necessary message senders, message receivers, and message types that are exchanged via the topic. Therefore, this component is provided by the EMI development platform.

The actual operation service provides a graphical user interface (GUI) to interact with its stakeholders. The GUI accesses a data component that provides storage, filtering, and aggregation of the data required for the operation service. This data can be metric data for the monitoring system, logging data for the logging system, or terms and their synonyms for the directory service. Additionally, the GUI utilizes a control component to send control commands to the EMI services. An example for such control commands is entering or exiting a maintenance mode, alive-checks, or reconfiguration of loggers.

5.8. Summary of the Technical Reference Architecture

This section presented our technical reference architecture for the core of an enterprise measurement infrastructure. We first presented an overview over all the layers and core services in an enterprise measurement infrastructure. Following the idea of microservices we split up the layers and services following separation of concerns. We then provided additional information and dedicated reference architectures to each of the layers and their services.

We started with the two integration layers. The most important one being the data integration and transport layer that houses the enterprise measurement data bus (EMDB). This bus provides the backbone for the data exchange in the data adaption and the calculation layer. The integration concept of the bus are puplish/subscribe topics to exchange EMDB messages between the data adapters and metric kernels. These messages use a generalization hierarchy to implement the satisfiability relation from our formalism.

We then moved to the calculation access layer. This integrates the visualization and the calculation layer using our enterprise uniform metric kernel access (EUrEKA) design. The core idea of EUrEKA is the specification for indicator access APIs that provide a uniform access to the indicators provided by the metric kernels. Additional services like the EUrEKA registry and the kernel description model ease working with EUrEKA.

After the integration layers we presented the reference architectures for the three core domain layers of our reference architecture. Following the data flow in an EMI we started with the reference architecture for data adapters. We presented our four data adaption patterns: push-forward, pull-forward, invoke-pull, and invoke-dump. Each pattern solves specific problems when integrating heterogeneous data providers in an EMI. For each pattern we presented the interactions between the data adapter and the data provider. Furthermore, we presented static reference architectures for the internal components of each type of data adapter.

The next section then presented the reference architecture for the heart of the EMIs: the metric kernels. We presented two different static reference architectures for different *sizes* of metric kernels: a monolithic design and a design with multiple components. Furthermore, we presented behavioral details for the two important functions of a metric kernels: processing messages from the EMDB and providing data via one of its indicator access APIs.

The following section completed the reference architecture for the EMI core layers with the reference architecture for visualization frontends in the visualization layer. We started with a classification and definition for the different types of visualization frontends. Similar to the metric kernels we then presented the static reference architecture for the visualization frontends as well as a discussion on the dynamic aspects including some of the optional components.

The last section then presented an architectural style to integrate the operation services and the EMI services with each other. The style proposes an agent-based integration between the different services. The following chapter will use this style to define the reference architectures for the different operation services.

6
Operation Systems and Services

An enterprise measurement infrastructure, as described in the technical reference architecture above, represents a large distributed system with a broad variety of different loosely coupled services. A key success factor to successful developing, operating, and managing such systems is solid monitoring and logging support. Similar to an EMI itself, these two systems represent small measurement systems with additional controls. Therefore, we start the description of each of the two systems with a list of information needs from the particular stakeholders (measurement customers). The stakeholders are mostly developers, metric experts, and operators. The reference architectures for the systems follow the architectural style defined in section 5.7 above.

Integrating heterogeneous data provider into one measurement infrastructure requires technical integration aspects as well as functional integration aspects. One of the functional integration aspects that is most frequently required is the integration of different terms for the same concepts in different data providers. To ease this task we propose the use of a *Directory System* that provides EMI services with different directories to lookup terms for given synonyms. Including this system in an actual EMI is optional but we found it to be very useful to lookup unique entity of measurement names or to reduce unnecessary complex status models.

As described above each of the following sections start with the requirements for the presented system. We then present the reference architectures for each of the systems. We start with the description of the monitoring system in section 6.1. Section 6.2 will afterwards provide the requirements and the reference architecture for an EMI specific logging system. After these two mandatory services we will introduce the optional directory system in section 6.3. We do not provide an overall summary at the end of the chapter as usual because the systems are too different and each section already summarizes the important points for each system.

6.1. Monitoring System

Monitoring and centralized control of the services is essential for successful operation of a measurement infrastructure. This is particularly true for a distributed system that contains a large number of loosely coupled services like EMIs based on our reference architecture. The job of the operator is to ensure that the measurement infrastructure and all of its services operate within their operational boundaries. Therefore, she needs a consolidated view on the status of all services in the measurement infrastructure. This status includes a lot of performance metrics on the services. Thus, the monitoring system is also a metric-based measurement system. The infrastructure operators act as the metric customers of this system. The development of the monitoring system therefore follows our metric systems development process model presented in part III. Following this process model the following section presents a list of information needs that guide the design of the monitoring system.

6.1.1. Information Needs Satisfied by the Monitoring System

The following list of monitoring information needs from measurement infrastructure operators further specifies the rough needs for the operators provided in section 9.3.5. These information needs are condensed experiences from our field studies. Most of them are also backed-up by existing literature for metric on service oriented architecture. For example Rud et al. who investigated different resource metrics for service oriented architectures [RSD07].

We differentiate between *live* and *statistical* information needs. The live information need reflect the current status of a service and is the basis for immediate actions. On the contrary, statistical information needs are based on statistical calculations on certain aspects of the services. Therefore, most of the time they do not require immediate actions from the operators or developers but stear the long time development of the EMI. The live information needs apply to all types of services. For the statistical information needs we provide dedicated subsections for the different types of services as well as universal needs.

Live Information Needs

As described above, these information needs trigger immediate actions from the operators. Therefore they need to be easily accessible by the operator and presented all the time.

What is the operational state of a metric service? This is the most crucial information for an operator. If a metric kernel goes into unplanned maintenance state or offline then the operator needs to do everything she can to get the service back online. We recommend to at least differentiate between the operation states: online, maintenance, offline or unknown.

In our field studies we also discussed the following operation states:

out-of-sync The data at the metric service is not the current (last) data from the data provider. This may happen if a service went offline during data updates from the data provider or when the updated data did not make it over to the EMI.

calculating This is a state specific to metric kernels when the calculation of a metric takes a considerable amount of time. Typically the arrival of new data triggers a new calculation. Until the calculation is finished the metric kernel is not able to show the current data.

heavy load The metric service receives a lot of messages or produces a lot of messages near the limit of the specified behavior. This is an indicator for metric operators to maybe move the metric service to a dedicated server or to initiate an improvement to the message handling of this service.

partially offline The metric service is still responding but some of its parts (most likely databases) are not responding. This will most likely lead to an "out-of-sync" state if not handled immediately.

It is also important to visualize "out-of-sync" and "calculating" to the metric customer and not just in the monitoring service. This helps the metric customer to judge the data and she is not fooled into wrong conclusions due to old data.

Furthermore, the architects, operators, and metric experts need to discuss and specify further states during the design of each service when instantiating (parts of) the references architecture. Operation states are, therefore, included in our template for the *design document* which defines the design for a development increment (see section 11 and section B.3 for further details).

Since when is a metric service not online? If a metric kernel leaves the online state (goes offline or in unplanned maintenance) then the most important information for the operator is the time at which the metric kernel left the online state. Using this the operator can filter the log or other information sources to find the cause of the problem.

What is the current value of important performance indicators? Each service needs to define its own performance indicators and their boundaries. The boundaries, however, can also depend on the actual measurement infrastructure and need to be defined for each service before staging it to the production environment. For example the upper boundaries for the message load of a service (number of incoming + outgoing messages to the EMDB) or the indicator load (number of requests via the indicator access APIs) depends on the node that hosts the service and its capabilities. Smaller hosts obviously reach their performance maximum earlier than larger hosts.

These information needs also need to be satisfied for sub-services an related services of an EMI or operation service. For example most services use a database or persistence service of some kind. Most of the time if the database is offline then the service is not able to perform its tasks. Important performance indicators for databases include the current number of database transactions of that service and the fill status of the database (number of items and/or disk space) with their specific current change rates for a given time frame.

An additional important information need for metric experts on all metric services is: "How is the sync state of the metric service?" The monitoring GUI can for example provide a list with all the entities of measurement for which the metric service is out-of-sync as an answer to this question. If the metric service is in-sync for all entities of measurements then the GUI should provide a corresponding information[1] to ensure the metric expert that the sync-tracing mechanism are operational.

Statistical Information Needs for all services

Most of the services in an EMI connect to the EMDB. Therefore most of the statistical information need is related to the reception and sending of messages. All the information needs should be available for different time scales and different time frames. For most of these we advice to provide: this hour, last hour, this hour yesterday, today, yesterday, this day last week, this week, last week, this month, last month, and all. If they are configured for *today* or shorter then these information needs almost become live information needs. Therefore, the time scales and time frames need to be added to the variability model of the respective metrics. The monitoring GUI can either provide a selection for them or provide a dedicated view for each of the scales and typical frames. The GUI can also offer combined views to provide better overviews; for example: this hour, last hour, and this hour yesterday combined on one GUI. Furthermore, we can provide additional statistical operations on these to determine the mean, median, or any other percentile distribution of the values[2]. These investigations can provide additional insides into causes for performance problems or optimization potential.

To judge the overall performance of a service the operator requires the consolidated message counts. The two most important information needs are therefore: "How many messages overall did it send?" and "How many messages overall did the service receive?". To find the cause of performance problems the monitoring system should also provide more detailed information. The two information needs for this are: "How many messages of what message type did it receive?" and "How many messages of what message type did it send?".

Statistical Information Needs for Data Adapter

The data adapter connect the measurement infrastructure with the data provider. They are crucial for up-to-date information in the visualization frontends. The

[1] For example: "The metric kernel is in-sync for all entities of measurements and has all current data."
[2] Always investigate whether the required statistical operation is allowed for the scale of the given metric!

push-forward data adapter pattern particularly requires constant load monitoring. The inversion-of-control like interaction between the data provider and the data gateway requires that the data gateway is always available. Otherwise, data loss is inevitable and the data adapter will get out-of-sync. To ensure the availability of the data gateways the operator needs to carefully watch and analyze the key performance indicators. The following information needs describe the foundation for the most important performance indicators.

How much data was adapted by an adapter in a specified time frame? This provides important information on the load of a data adapter. The data can either be represented by counting logical units (e.g. number of tickets) or data size (e.g. megabyte).

How much data adaption was rejected in a specified time frame? The data rejection of a data adapter can be a good indicator for the data quality problems in a data provider or the quality of the connection between the data provider and the data adapter. If the rate or the absolute number is to high then the operator needs to utilize the logging system to investigate the reason for the rejection.

How much data was provided by each data provider of the adapter? A data adapter can be responsible for the integration of multiple data provider as we discussed in section 5.4. Therefore, a data provider that spams a data adapter with information can block calls from other data providers resulting in data loss, similar to a DOS attack. An important indicator for this is the number of calls for a specific data provider. If this number gets too large for certain providers then the operator may provide several physical deployments of the same data adapter and reroute the calls from the data providers.

Statistical Information Needs for Metric Kernels

The metric kernels are heart of an EMI. Therefore it is important to ensure that these are operational. The message related information needs described above and the service state of the databases of the metric kernels are the most important information needs. Additionally, operators are interested to know: “How many EUrEKA requests are performed through what indicator access API?”. If they notice a high load on a specific indicator access API this maybe need to be located to an additional metric kernel. Alternatively, the metric kernel needs to be deployed multiple times and the API needs to be accessed via a load balancer.

Contrasting the information needs of the operator, the metric expert is interested in functional questions related to the definition and design of the metrics. Answers to the following information needs can help to optimize a metric portfolio. Each of these information needs need to be configured with a given time frame.

What metric ids where requested how many times? An answer to this question is to use a counter for the number of calls to each metric identifier. If a metric kernel implements a large number of metrics then a good answer is to just show the top-5 and bottom-5 metric ids. The metrics that are not accessed are potential candidates for removal from the metric portfolio in one of the next development increments. This reduces the complexity of the metric portfolio and the metric kernel. If the load of the metric kernel is to high then it is a good idea to move the most used metrics to their own metric kernel in one of the next development increments.

How often is what variability point configured? Similar to the question above this just requires counting the usage and the configuration of each variability point. If the variability model defines to many variability points then, similar to above, the top-5 and bottom-5 variability points are sufficient. If a variability point is never used it can potentially be removed similar to unused metrics. This again reduces the complexity of the metric portfolio and the metric kernel. If one of the variability point is always configured with a similar variability configuration then maybe this requires to split the metric and define individual metrics for the top most used variability points to eases the configuration of the metrics.

How many requests did result in no data? A sudden raise in the number of requests that result in no data is an indicator for a problem in the integration of a data provider or configuration errors in the visualization frontends.

Statistical Information Needs for Visualization Frontends

Similar to the metric kernels the operators are only interested in one additional information need related to the load of the frontend: "How many users are interacting with the visualization?". Optimally, this information need is satisfied using a chart or list for the usage. These should be configurable to: hourly, daily, weekly, and monthly. For example: hourly would produce a table or chart for the last values of the last 24 hours. Because this is an aggregated value the GUI should also provide the selection for the aggregation operator: `max`, `min`, `avg`, and `abs`.

Most of the information needs related to the visualization frontends come from the metric experts. They like to optimize the metric portfolio and assist the metric customers. All the following information needs focus on M^2 dashboards. Analysis tools are typically used by the metric expert or other experts in the company. Consequently, they should know what they are doing and the metric expert does not need to assist them as much. Hence, she does not require according performance indicators. If an analysis tool indicates a certain problem then typically this triggers the development process to define specific metrics and monitors to analyze the problem constantly in one of the dashboards. See section 12.2.3 in the operation phase of our process model for further details. To get an overview over the utilization of the visualization frontends they stated the following needs: (Similar to above, all these need to be configured for a given time frame)

How many M^2 dashboards are configured? The number of M^2 dashboards that are configured in an M^2 dashboard tool is often used as a normalization. For example to calculate the average number of monitors per M^2 dashboard or the relative amount of custom configured M^2 dashboards.

How many monitors are placed in the M^2 dashboards? This value also provides a basic indicator for the customization of a dashboard. The monitoring should provide the basic aggregation levels: min, max, avg, and abs to completely satisfy the need.

How different are the M^2 dashboards configured? The metric expert can provide M^2 dashboard best practices to provide the metric customers with ready to use M^2 dashboards. This need helps the metric expert to find new best practices in the currently configured M^2 dashboards. The configuration differences should be provided on the basis of monitor clusters and on similarity metrics on the M^2 dashboards. A monitor cluster is a set of similar monitors. A good similarity metric between two M^2 dashboards for example is the number of actions (add, modify, and delete monitors as well as M^2 dashboard grid resizing) required to transform the one M^2 dashboard into the other one.

Some M^2 dashboard tools provides an explicit templating mechanism for dashboards and monitors. The metric expert then also requires information about their usage to optimize the template basis.

How many custom dashboards are configured? If too many custom dashboards are configured then this is an indicator for missing templates. The metric expert should analyze the custom dashboards for example via the configuration difference indicator from above and try to extract new templates.

How many custom monitors are configured? Similar to the need above, this is a means to indicate missing monitor templates.

What is the template usage top-5 and bottom-5? The bottom-5 provides templates that can potentially be deleted if they are not used frequently. The top-5 provides a list of best practices that can be presented to new employees.

This concludes our list of monitoring information needs from measurement infrastructure operators and metric experts. Similar to other measurement systems this list needs to be continuously evaluated on the actual EMI and extended if necessary. The following sub section will provide a reference architecture for the monitoring system that implements the style presented above in section 5.7.

6.1.2. Monitoring System Reference Architecture

The reference architecture for the monitoring system implements the agent-based integration style presented above in section 5.7. This section will present the instantiation of the style in a static reference architecture. Furthermore, we provide additional information on the alive check and service discovery dynamics as well as the dynamics of the performance indicator calculation and transformation[3]. The communication between the monitoring client and the monitoring service are based on the request-reply enterprise integration pattern [HW03a]. The section starts, however, with the static reference architecture for the monitoring system.

Static Monitoring System Reference Architecture

As described in section 5.2.1, the monitoring system uses the `EMI.monitoring` topic to exchange information between the agents. We do not provide an additional overview diagram[4] for the complete monitoring system but focus on the detailed reference architectures for the monitoring client agent and additional details for the monitoring service.

Figure 6.1 provides an overview over the static reference architecture for the monitoring client agent. The client agent connects to the monitoring topic (`EMI.monitoring`) using dedicated `Monitoring Message Sender` and `Monitoring Message Receiver` components which exchange `Monitoring Messages`. These messages can be extended depending on the specific communication required in an EMI instance. Default messages include: `Alive Check Request`, `Service Details`, `Service Shutdown Info`, `Performance Indicator Request`, `Performance Indicator Response`, and `Monitoring Event`. We describe additional details on the semantic and behavior that are associated with these messages in the sub sections below.

The central `Monitoring Client Agent Controller` orchestrates all the interactions in the agent. Most importantly, this controller uses the `Performance Indicator Source Registry` to produce performance indicators when answering performance indicator requests. The registry holds a number of `Performance Indicator Sources`. These sources implement the performance indicators required to answers the information needs from above. The result of a calculation of a performance indicator is then wrapped in a specific `Performance Indicator Representation`. These representations are stored in the response and transported to the central monitoring service which visualizes them accordingly. The representations and indicator sources can be further specialized to implement specific representations for an EMI instance.

The monitoring client agent also provides an interface for `Monitoring Service Access`. This interface can trigger specific actions in the client agent. For example the sending of a monitoring event or reporting the startup or shutdown of a service.

[3] Additional details to the concepts presented in this section can be found in the thesis of Ahmet Yüksektepe [Yük13].

[4] See section 5.7 and figure 5.38 and figure 5.39 for an overview on the integration between the monitoring service and the EMI services.

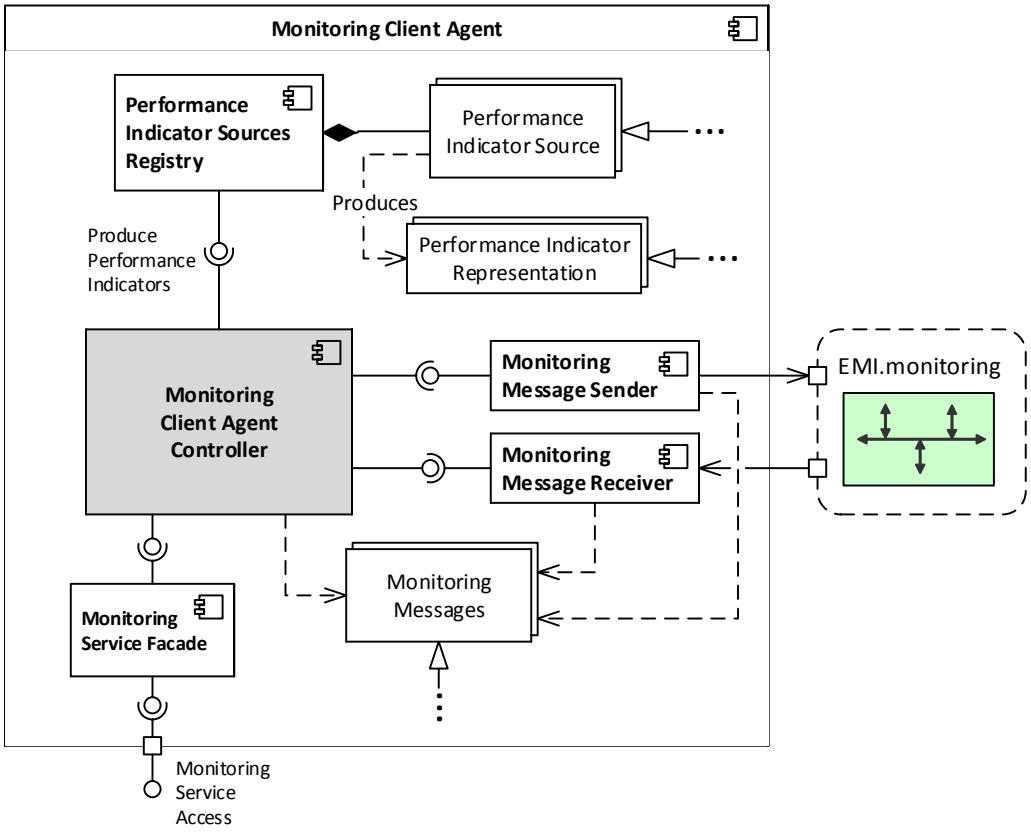


Figure 6.1.: Static reference architecture for the monitoring client agent

This interface also provides access to some of the indicator sources. For example we recommend to provide methods to calculate the message-based performance indicators: `onMessageReceive(message)` and `onMessageSend(message)`. The controller can then delegate the calls to the registry which delegates them to the message-based performance indicator sources. These then increase their counters or extract size information from the messages.

The static reference architecture for the monitoring service is very similar to the architecture provided in the style. Therefore we do not provide an additional component overview. In accordance with the architectural style, the monitoring service also includes a monitoring client agent. Hence, the performance indicators for the monitoring service itself use the standard mechanisms to provide performance indicators about itself rather then a shortcut. This produces additional communication overhead but reduces the overall complexity of the monitoring service because it only needs to implement the standard way.

The monitoring system also provides operation state information to other services. This is particular important for the EUrEKA consumers (see section 5.18). These need to check if their requests can be answered before actually querying an API and running into a timeout for offline or maintained services. The following listing 6.1 provides the methods of the service operation status API using our API specification language (see section 3.2 for additional details):

Source Code 6.1 API specification for the service operation status API

```
interface ServiceOperationStatus

  mandatory method getOperationStatusFor
    mandatory parameter serviceName : String
    returns OperationStatus

  optional method isServiceAvailible
    mandatory parameter serviceName : String
    returns Boolean
```

The method `getServiceStatusFor` returns the actual operation status of the service. The optional method `isServiceAvailible` is more convenient for the check. However, it returns true if the operation state of the given service is not `offline`, `unknown`, or `maintenance`.

The following two sub sections provide additional information on the dynamics between the monitoring service and the monitoring client agents as well as details on the performance indicator provisioning.

Dynamic Aspect: Alive Check and Service Discovery

The most central goal of the monitoring system is to provide information on the operational state of the services. To report this, the monitoring service periodically triggers alive-checks. In our EMIs we choose to fire it every 10 seconds. An exception to the alive-check are services that are currently in `planned maintenance` operation state. This state indicates that someone is working on the service. Hence, the monitoring system can ignore it.

One alternative for the implementation of the alive-checks would be to check the aliveness of all services from the monitoring service. The service would iterate over the list of all of its services and tries to get an answer from each service. The problem with this, however, is that the iteration is synchronized. Hence, if a service failed and does not answer the request the monitoring service needs to wait for a timeout the identify that a service is not alive. This would slow the alive-check significantly and hence the trigger interval needs to be increased accordingly. The alive information, however, is critical and the interval should be capt as low as possible.

Therefore, we recommend an asynchronous alive-check. Figure 6.2 provides a UML activity diagram for this asynchronous behavior of the monitoring service and monitoring client agent during an alive-check. After the alive-check is triggered the monitoring

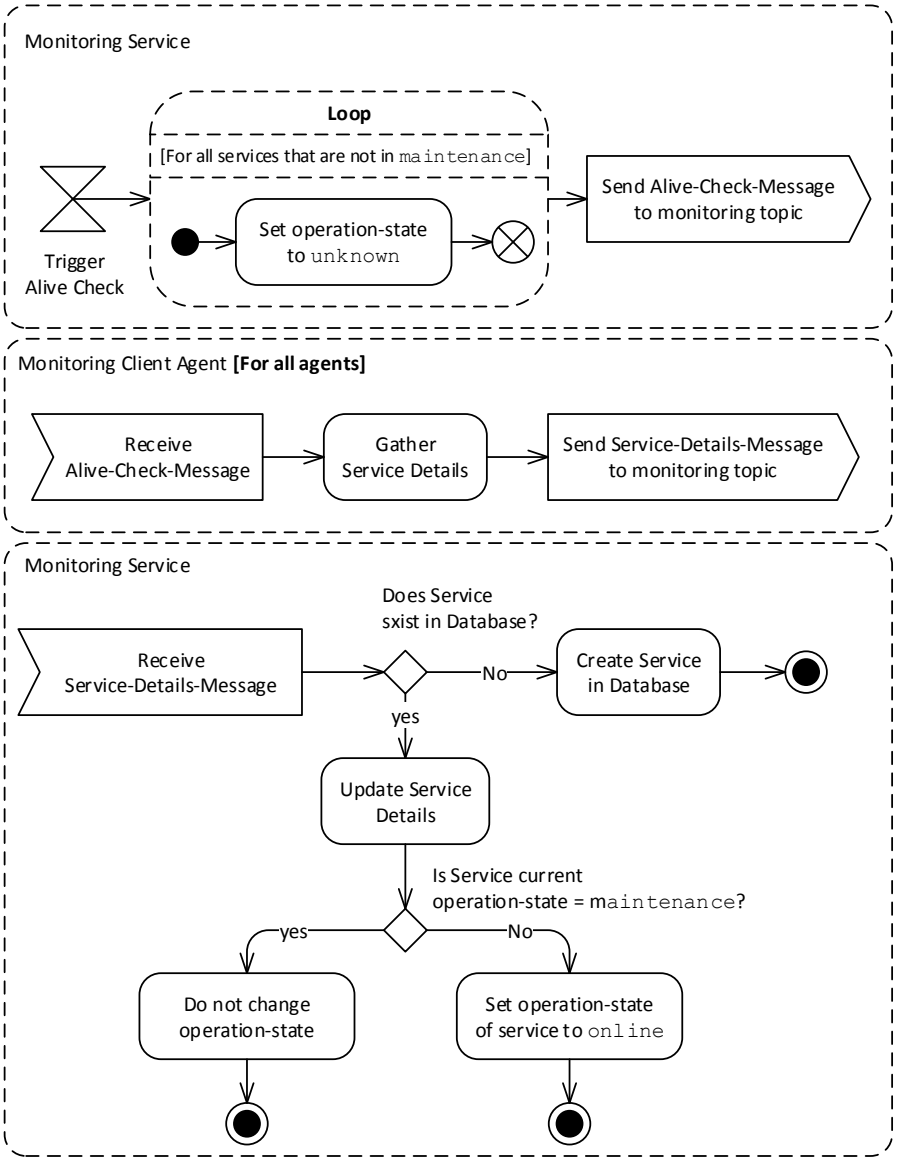


Figure 6.2.: UML activity diagram for the behavior of the monitoring service and monitoring client agent during an alive-check.

service first sets the current operation state of all services to `unknown`. After that the service sends out an `Alive-Check-Message` to the monitoring topic that forces all monitoring client agents to answer with their service details. Therefore, a Service that does not answer will remain in `unknown` state. We choose to set it to `unknown` state rather than `offline` because we do not know for certain that the service is offline. It may just perform a calculation heavy operation that delays the answer to the alive-check.

The service only enters offline-state after the monitoring service receives a Service-Shutdown-Info-Message from the service. This message needs to be send during the shutdown of the service. This can be included in the destructor for the monitoring client agent component for example or in an explicit shutdown hook provided by the component infrastructure.

After sending the alive-check-message, the monitoring service asynchronously receives Service-Detail-Messages from the monitoring client agents. This message is not only send as an answer to alive-checks but also send during service startup. Therefore, the monitoring service first needs to check if the service already exists in its data base. If not it creates it with the provided service details. Thus, this behavior conveniently implements the service discovery function. If the service is already known to the monitoring service then it updates the details of the service with the provided information in the message for example to update the description of the service or its detailed operation state. If the service is currently being maintained then the monitoring service does not change the operation state. Otherwise, it sets the operation state to online.

This concludes the specification of the alive-check behavior and service discovery. The next section will provide details on the performance indicator provisioning.

Dynamic Aspect: Performance Indicator Provisioning

Performance indicators answer the information needs from above. Hence, the performance indicators need to be implemented in the monitoring client agents because stakeholders have different questions for different types of services. The actual list of performance indicators provided by a service is defined in the design phase of our process model (see section 11.2.1 for additional details).

The performance indicator provisioning starts with an interaction of a stakeholder in the GUI of the monitoring service. The stakeholder likes to answer her information needs regarding a specific service. The monitoring service then, similar to the alive-check described above, sends out a Performance-Indicator-Request-Message for the specific service. The message receiver in the monitoring client agents catch the message and check if it needs to provide performance indicators. If so then the message receiver triggers the calculation in the controller. This delegates the production of the indicators to the Performance Indicator Source Registry.

Figure 6.3 provides a UML sequence diagram for the production of the performance indicators by the performance indicator source registry. The registry loops over all its performance indicator sources and triggers the buildIndicator method on each source. In an implementation this method would be a template method on a performance indicator base class. The method first calculates the actual indicator values based on its internal storage. Then it instantiates an appropriate Performance Indicator Representation and provides this with the calculated value. The indicator is then returned and added to the result list of the registry. When all the indicators are build then the registry returns the result list with all the indicator (representations).

The list is then wrapped in a Performance-Indicator-Response-Message by the controller of the monitoring client agent which is send using the monitoring message

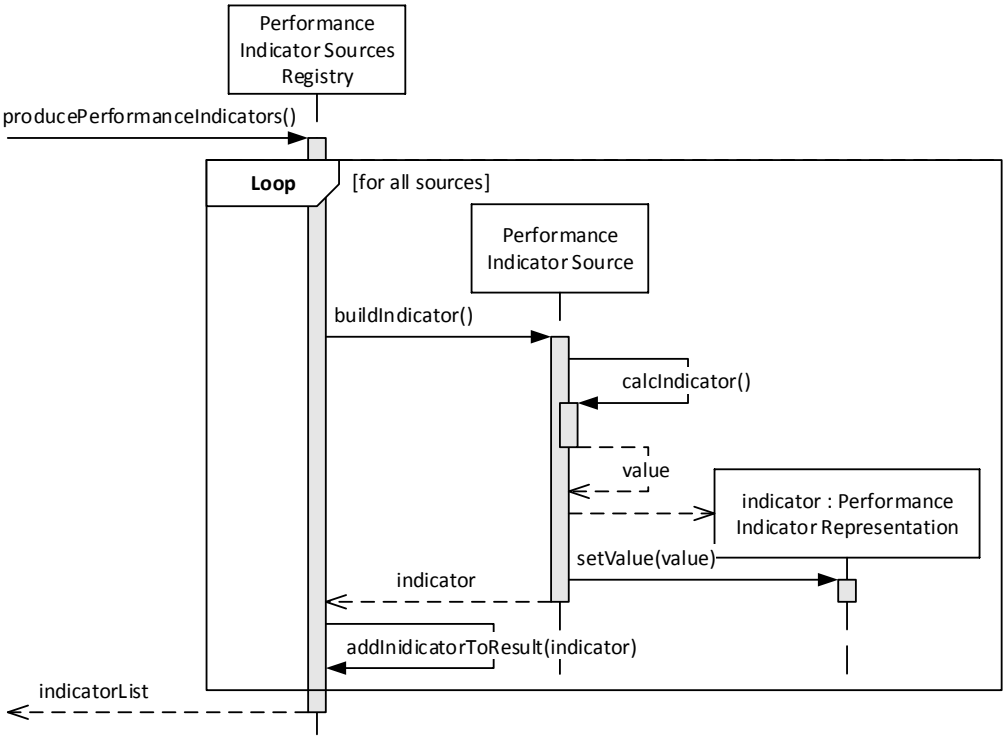


Figure 6.3.: UML sequence diagram for the production of the performance indicators in a monitoring client agent.

sender. The monitoring service catches the message using the master agent and updates the GUI with the visualizations for the indicator representations in the list.

Depending on the implementation technology of the monitoring client agent the key behavior using the indicator sources can be delegated to existing libraries. The Metric library for example (`https://dropwizard.github.io/metrics/`) provides the required mechanisms related to indicator sources and indicator calculation for java-based implementations. However, we recommend to implement different performance indicator representations based on the actual representation needs; for example: tables, charts, or plain text. Furthermore, we also encourage the custom implementation of the registry and indicator sources and just hooking in existing libraries. Thereby, the indicators sources can easily be extended with special sources for specific services (if needed).

6.1.3. Monitoring System Summary

This section provided a brief introduction of the the monitoring system. We started with a large list of information needs from various stakeholders to the monitoring system. We differentiated live information needs and statistical information needs. They describe important information for the operator and the metric expert. The most important one for the operator is the operation state of each service.

Statistical information needs are only answered on request. They include performance indicators for each service. We provided different performance indicators for each type of EMI service as a guideline for the actual definition of all monitoring information needs related to each service when instantiating the reference architecture.

We then provided key aspects of the reference architecture for the monitoring system. The statical reference architecture focused on the monitoring client agent and its components. Furthermore, we also provided the API specification for the operation state API of the monitoring service. This API is crucial for the smooth operation of the EUrEKA consumers; hence, the visualization frontends.

We also provided additional details on the request-response based dynamics in the monitoring system. First we presented the alive-check and service discovery behavior. This uses asynchronous mechanics based on service-detail messages to discover services and reset their online state. We also provided additional details to the calculation of performance indicators inside the monitoring client agent.

The goal of the monitoring system is to provide operators and metric experts with quick answers and overviews over their information needs. However, if a performance indicator hints a problem or a specific aspect needs to be investigated then they need more insides into the services. This inside is provided by the logging system which will be introduced in the following section.

6.2. Logging System

This section provides central information needs and key aspects of the reference architecture for the logging system. Following the definition from Chuvakin et al. we define a Log as [CSP13]:

> At the heart of log data are, simply, log messages, or logs. A log message is what a computer system, device, software, etc. generates in response to some sort of stimuli. [...] First off, the typical basic contents for a log message are the following: Timestamp, Source, [and] Data.

A large distributed system that contains a high number of loosely coupled services like EMIs based on our reference architecture requires a central logging mechanism and a logging infrastructure [KS06]. Logs provide inside information required for a variety of different tasks related to the development and operation of an EMI. The following section provides a brief overview over the central information needs from the developers, operators, metric experts, and metric customers. After that we present key aspects of the reference architecture for the logging system[5].

6.2.1. Information Needs Satisfied by the Logging System

This sub section provides information needs that are answered by the logging system. However, the logging system addresses a large variate of stakeholders (developers, operators, metric experts, and metric customers). Therefore, we differentiate between technical logging and functional logging. Consequently, the GUI of the logging service needs to provide different interfaces (or sub-interfaces) for the different types of logs. Furthermore, the logging service can also provide functional logs via an API, which can be queried by the visualization frontends to retrieve (functional) logging information relevant for a particular M^2 dashboard.

The most important requirement for the logging system is to provide a centralized logging service for all the distributed services in an EMI. This logging service can help to increase development speed and development quality because developers can easily access the log of the complete EMI. Also, this greatly eases the root cause analysis for problems in the production environment by operators and developers. However, the strength of the microservice architectural style is that each service can freely chose its technologies. This leads to a lot of difficulties when realizing a centralized logging system because the logging client agents need to be compatible with all the different technologies.

This section can only provide very few actual information needs because logging is very closely related to the actual design of the services. Consequently, the actual logging information needs need to be defined when designing the behavioral details of a service (see section 11.2.1). These needs then need to be evaluated by the stakeholders when evaluating the design of the service (see section 11.2.3).

[5] Additional details to the concepts presented in this section as well as valuable information on the actual implementation of the reference architecture in a Java environment can be found in the thesis of Jan Döring [Dör14].

Hence, this section only provides very basic and generic information needs. We start the overview of the information needs with the technical information needs from developers and operators.

Technical Information Needs

The technical roles (developers and operators) are mainly interested in technical information provided by the services. This information provides feedback from the service to the developers during the construction phase when the service is not integrated into a complete EMI. Technical information is also important for operators. They are responsible for successful operation of the measurement infrastructure. If the monitoring system shows a failure of a service or a strange behavior then the operators can check the technical log of the service to find the root cause of the problem.

Functional Information Needs

The main difference between functional logging and technical logging in an EMI is that functional logging information is always specific to a given entity of measurement. Thereby the information can easily be filtered to the relevant eoms for a specific user. This is particular important for metric customers who need to get additional inside information from the EMI upon specific actions. Typically they only require information from a very narrow set of entities of measurements. Hence, it would be hard for them to focus on the relevant information when they would see all log information for all entities of measurement.

The metric expert or even metric customers need to be able to investigate details of data rejection by the data adapter. Data rejection will typically lead to a data adapter, and consequently all related metric kernels, to enter out-of-sync state. Therefore, the problems related to the rejection need to be fixed as fast as possible. Hence, a specific functional logging information need related to the data adapters is “Why was certain data rejected?”. To answer this need the data adapter needs to log the rejected data and rejection reason.

Another specific functional logging information need from metric experts is related to the data requested by EUrEKA consumer. The monitoring system provides information about how many requests return no data. If this indicator suddenly changes or the number is high then the metric experts needs to investigate further details. Therefore the EUrEKA consumer or the metric kernel need to answer the need: “What EUrEKA requests resulted in no data?”. To answer this they need to log the URL or identifier of the API, the metric identifier, and the entity of measurement as log data.

As stated above the actual logging information needs heavily depend on the design of the specific service. Therefore we do not provide further information needs to encourage the specific definition. The next section will provide an overview over the key aspects of the reference architecture for the logging system.

6.2.2. Logging System Reference Architecture

The reference architecture for the logging system implements the agent-based integration style presented above in section 5.7. This section will present the instantiation of the style as well as additional information on the static reference architecture and information about the behavior when reconfiguring and discovering a logger.

As described in section 5.2.1, the logging system uses the `EMI.logging` topic to exchange information between the agents. We do not provide an additional overview diagram[6] for the complete logging system but focus on the detailed reference architectures for the logging client agent. Most technologies already offer dedicated logging support. Therefore, the goal of this reference architecture is to integrate the existing loggers with our centralized logging service.

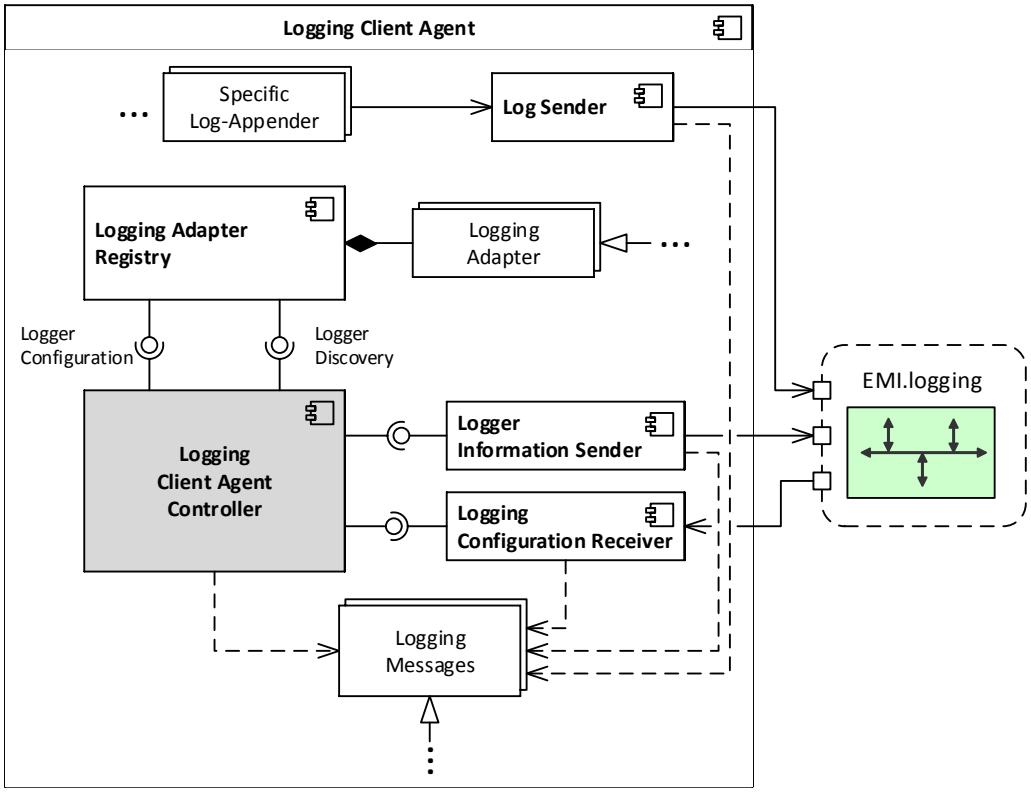


Figure 6.4.: Static reference architecture for the logging client agent

Figure 6.4 provides an overview over the static reference architecture for the logging client agent. The client agent connects to the logging topic (`EMI.logging`) using the `Log Sender`, `Logger Information Sender` and `Logging Configuration Receiver` components which exchange `Logging Messages`. These messages can be extended depending on the specific communication required in an EMI instance.

[6] See section 5.7 and figure 5.38 and figure 5.39 for an overview on the integration between the logging service and the EMI services.

Contrasting the static reference architecture of the monitoring system, the log sender does not connect to the central `Logging Client Agent Controller` but directly to the `Log Appender`. Log appender can be configured to listen to specific log messages produces by the service. Different technologies require different appenders. Therefore, the logging client agent may require to provide different log appender to fit the specific technologies.

The `Logging Client Agent Controller` orchestrates all other (non log related) interactions in the logging client agent. It utilizes the `Logging Adapter Registry` to configure and discover loggers provided by different technologies. Each technology requires its own `Logging Adapter` which is then registered with the registry. The result of the logger discovery, all loggers of the service, is send to the logging service using the `Logger Information Sender`. The `Logging Configuration Receiver` listens for new logging configurations. Fitting configurations are delegated to the controller which passes them to the registry. The registry then finds the suitable adapter which applies the configuration on the logger.

The actual implementation of this reference architecture can become quiet complicated; depending on the technologies used. However, we believe that the static reference architecture provided above and the rough description of the behavior is enough to understand the basic ideas of the design. Therefore, we do not provide additional information on the dynamic aspects of the logging system. Please refer to the additional sources provided in the introduction for further details.

6.2.3. Logging System Summary

This section provided a brief introduction of the reference architecture for a centralized logging system for EMIs. We started with a brief overview of the logging information needs. In there we distinguished between functional and technical logging. Each of which addresses different stakeholders. Most importantly, we emphasized the importance of defining the specific logging information during the design of the actual services.

We then provided key aspects of the reference architecture for the logging system. We focused on the statical reference architecture for the logging client agent. With this we also briefly discussed the responsibilities of specific log appender. Additionally, we provided short descriptions of the logging configuration and logger discovery mechanism.

This concludes our discussion of the two most important operation services: Monitoring and Logging. The next section will introduce the optional directory services which eases the integration of heterogeneous data in an EMI.

6.3. Lookup System (optional)

The Lookup System implements a generic service to lookup terms for synonyms. Even though it is optional, we recommend to include it because it drastically reduces the setup and maintenance effort of an EMI. The centralized directory service of the directory system provides a single place in the EMI where operators or metric experts are able to specify all the synonyms for actual terms. The following sub section introduces the goals and requirements for the directory system more thoroughly using user story oriented use cases. After that we provide key aspects of the reference architecture for the directory system[7].

6.3.1. Use Cases and Requirements for the Lookup System

The two most important use cases[8] are the lookup of unique entities of measurements and the reduction of enumerations. The following sub sections provide additional details for the two use cases.

EOM Unification

The (functional and technical) integration of heterogeneous data providers in an EMI raises a huge amount of challenges. One of the functional challenges is different naming of important properties; most importantly the entities of measurements. In our field studies we observed that the names for the same entity of measurement in different systems are often different. For example in an issue tracking system a project may be called "Project 0815", in the version control system it is known as "/cvsroot/projects/0815", and the accounting system only know it as "P0815". However, a project manager would like to satisfy issue related, configuration related, and accounting related information needs in a single M^2 dashboard without looking up all the different identifiers. Even more important metrics that require the input from different data provider need to be able to actually calculate the metrics without too much configuration on them. For example the metric "Average costs per bug" requires information from the accounting system and the issue tracker. Hence, all the different identifiers (synonyms) for one project need to be identified as the same project (term) inside the EMI. Our solution is therefore to use the directory service to define one unique identifier for the eom and register all it's synonyms with this identifier.

Enumeration Reduction

Another huge functional challenge when integrating heterogeneous data providers in an EMI is the reduction of enumeration values for example issue states. Issue tracking systems are required to reflect the underlying supported processes.

[7] Additional details to the concepts presented in this section can be found in the thesis of Nick Russler [Rus13].

[8] These are not *use cases* as described in the UML. We provide a more informal description of the cases of using our system. Hence, these uses cases resemble user stories rather than formal UML use cases.

This typically results in a large variety of issue states (e.g. “issued”, “in check from customer”, “requires customer feedback”, “requires feedback from customer management”, and so on ...). All these states are important to track the actual issues in the real process. However, they are not necessarily important to someone who is just interested in a course grained analysis.

On a very basic level a project manager, for example, is just interested in the number of “Open” and maybe “Closed” issues. Hence, for this counting metric most of the fine grained states are just synonyms to “Open”. However, these states should not be *hard coded* into the metric because the actual EMI may integrate multiple issue trackers and the states change over time. Therefore, a metric expert can use the directory service to define all the synonyms for the function states “open issue” and “closed issue”. The metric then uses the directory system to translate the actual status into the functional status.

Requirements

The obvious requirement for the directory system is to store terms for given synonyms. In order to support the different use cases the directory system should provide different *directories* in which to store the terms and their synonyms. Hence, a service can use the “EOM Directory” to lookup the actual term for an eom synonym and then use the “Issue Status Directory” to lookup the function issue status for the actual status of a given issue.

Following our style from section 5.7 the directory system should also provide a central directory service for the definition of terms and their synonyms. The directory system can also provide synonyms to the service automatically for example when a EMDB message contains a new/unknown eom. Hence, the metric expert or operator just needs to perform the term definition and assign the appropriate synonyms to the terms. However, the service should also support to manually add a (bunch of) synonym(s) in order to setup the directory system with all term-synonym relations before a message is send.

This concludes the use cases and requirements to the directory system. The following section will introduce key aspects of the reference architecture for the directory system.

6.3.2. Directory System Reference Architecture

Similar to the monitoring system and the logging system, the reference architecture for the directory system implements the agent-based integration style from section 5.7. This section will present the instantiation of the style as well as additional information on the static reference architecture and information about the behavior when actually performing a lookup of a synonym in the directory.

Contrasting the Monitoring System and the Logging System we provide an overview of the complete Directory System in figure 6.5 because it differs slightly from the integration style. Most importantly, the Directory Service does not contain a Directory Client Agent and we do not require a control component. The directory service therefore only contains the Directory Service Master Agent which integrates the directory service with

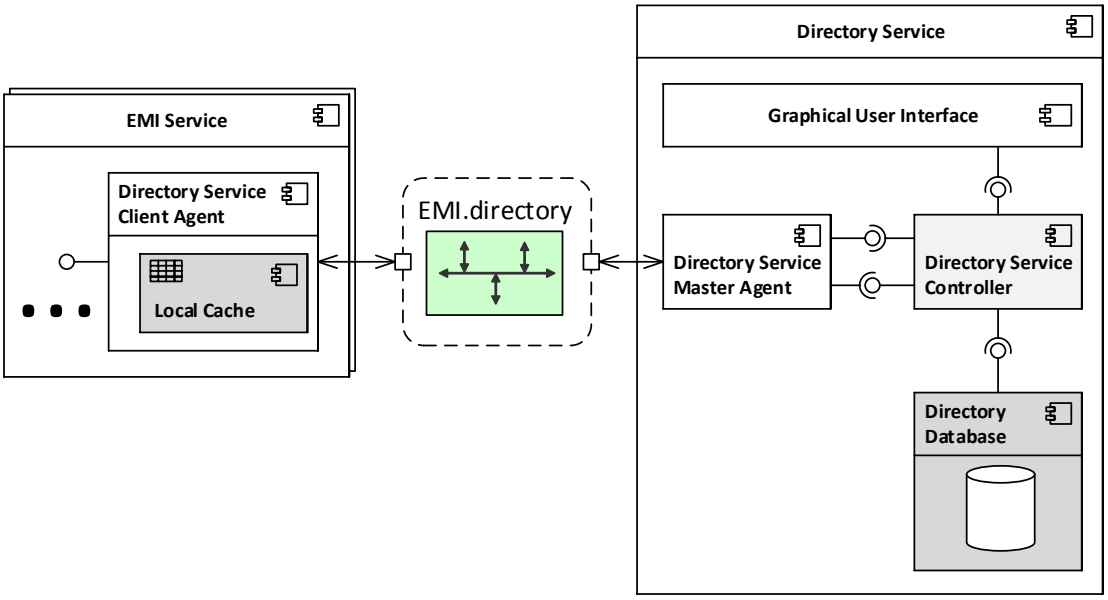


Figure 6.5.: Static reference architecture for the directory system

the EMI services. The master agent is closely coupled to the Directory Service Controller which orchestrates the control flow inside the directory service. Most importantly, the controller connects the agent and the GUI to the Synonym Database which stores the terms, synonyms, and their relations. The Graphical User Interface (GUI) of the directory service provides the web frontend to the metric experts and operators.

As usual the directory client agent connects to the topic using dedicated `Directory Message Senders` and `Directory Message Receivers` to exchange `Lookup Messages` (all three not shown in figure). Similar to the monitoring system, the directory system also utilizes the request-reply-pattern to implement the communication between the client agent and the master agent [HW03a]. Hence, the two most important lookup messages are: `Lookup-Request` and `Lookup-Response` which transport the request and the reply respectively.

We recommend to add a local cache to the directory client agent to speed up the lookup of known synonyms. Caching of previous lookup results prevents multiple costly request-reply-based lookups in the central directory service. However, this caching also increases the complexity of the management of synonyms and terms in the central directory service because all actions need to be published over the topic to update the local caches. The additional messages required for this are: `UpdateSynonymTermRelation` and `DeleteSynonymTermRelation`. However, actual lookups are fare more frequent than the (re)-specification of terms and their synonyms. Hence, it is justified to add complexity to speed up the more frequent action. Using the local cache also speeds up the message reception and calculation in metric kernels significantly because they need to perform a lookup of the eom every time they receive a new message.

Most importantly, the Directory Client Agents provides an interface to the EMI services to utilize the functions of the directory service. The following listing 6.2 provides the methods of this lookup interface using our API specification language (see section 3.2 for additional details). We will, as usual, provide additional details to the methods after the listing.

Source Code 6.2 Specification for the directory lookup interface of the directory client agent

```
interface DirectoryLookup

  mandatory method lookupSynonym
    mandatory parameter directory : String
    mandatory parameter synonym : String
    optional parameter timeoutInMiliSecounds : Integer
    returns  String

  optional method lookupSynonym
    mandatory parameter directory : String
    mandatory parameter synonym : String
    optional parameter timeoutInMiliSecounds : Integer
    returnsLookupResult

  optional method getSynonyms
    mandatory parameter directory : String
    mandatory parameter term : String
    returns List of String

  optional method getTermsInDirectory
    mandatory parameter directory : String
    returns List of String

  optional method registerSynonyms
    mandatory parameter directory : String
    mandatory parameter synonyms : List of String
    returns Void
```

The `lookupSynonym` method resolves the provided synonym to it's term in the given directory. Optionally, the method can contain a timeout parameter. If the request-reply-based lookup via the lookup topic exceeds the given timeout then the method will simply return the synonym. In our implementations we use a default timeout of 2000 milliseconds if no timeout is specified. The method can either return a String for the resolved synonym or a more complex `LookupResult`. This can provide additional information about the lookup together with the term that resolves for the synonym. The additional information can be used for logging or to visualize lookup results to stakeholders when configuring certain EMI services.

Optionally, the interface can provide a method to access all synonyms for a given term with the `getSynonyms` method. The list can, for example, be shown when configuring the eom of a monitor in a dashboard in order to provide information about all actual eoms that feed the monitor. Similarly, the optional `getTermsInDirectory` method can be used to, for example, show all registered eoms.

The optional `registerSynonyms` method allows EMI services to register new synonyms in a given directory. Thus, the manual specifications of synonyms by the metric experts or operators can be avoided, which saves time and prevents spelling errors.

We believe that the static reference architecture provided above and the rough description of the behavior is enough to understand the basic ideas of the design of a Directory System. Therefore, we do not provide additional information on the dynamic aspects of it. Please refer to the additional sources provided in the introduction for further details.

Directory System Summary

This section provided a brief introduction of the directory system. We started with a brief introduction to the two most important use cases of the directory system: Unifying entities of measurement and simplifying enumerations. From these we distilled high level requirements to the lockup system. Most importantly the need for different directories of terms and synonyms.

We then provided key aspects of the reference architecture for the directory system. We focused on the static reference architecture of the system. With this we provided a specification for the lookup interface that is used by the EMI services to interact with the directory system. Furthermore, we provided short descriptions of the behavior of the methods as well as a brief discussion on the benefits and weaknesses of using a local cache for the synonym-term-relations.

This concludes our discussion of the operation systems. The next section will extend our formalism from the foundations to the concepts used in the reference architecture.

7

MeDIC Reference Architecture Formalisms

As the psychologist Kurt Lewin once stated:

> There is nothing more practical than a good theory.

We embrace this by providing a formal basis to our, practically oriented, reference architecture for enterprise measurement infrastructures in this chapter.

We first provide a formalism for the service states of the services in an EMI. The state of the service is the most basic information that an operator requires of a service. The service states are therefore prominently placed in the GUI of the monitoring service. However, we require a solid formalism to define the semantics and behavior associated with a certain state. Therefore, we present a set-based formalism for the service states in the following section 7.1.

We already provided a formalism for our measurement data flow in section 2.3. However, the actual concepts of the MeDIC reference architecture are not addressed by the formalism. Therefore, we need to extend it to be able to formally describe and analyze an actual EMI. This enables us to investigate the termination of calculations inside an actual EMI. Additionally, we are able to investigate all possible measurements available in an EMI based on the data in the data providers. We call this the *reach* of the data in the EMI; technically we calculate the outer calculation hull. This extended formalism is provided in section 7.2.

The application of a formalism is often difficult due to its abstract nature. Therefore, is is very useful to provide examples of the application of the different concepts. We already provide some small examples with the definition of the formalism. However, a large example that uses all the different concepts and shows their application is very useful. Therefore we provide a very detailed example of the application of the formalism in section 7.3.

7.1. Formalism for Service States

Service states are a rough means to communicate run time information about the services. This information is a crucial indicator to metric operators and metric experts when they inspect whether the measurement infrastructure runs within sound parameters and acts as required. First of all, they should be accessible through the monitoring system. Additionally, visualization frontends should also be able to access the states to indicate them next to its monitor. The following two sub sections define the two most important states: "Maintenance" and "In-Sync with data provider". The actual EMI instance, however, can extend these states if required.

Some states can *ripple* through the different layers of the EMI. If, for example, a data adapter is out of sync then all the metric kernels that require measurements from this data adapter will also be out of sync and so forth. Hence, when reporting service states then the root cause (service) should be reported as well. This root can be used to filter specific information. For example if the metric application is not in sync with a specific data provider this information is only required for metric customers who require data from this specific provider.

Each service state $State_i$ is defined as a set of tuples of EMI services $\mathfrak{S}$ and identification depending on the needs of the operators or the detection potential for that state. For example:

$$State_{Example} = \left\{ (\mathfrak{S}, \{ident_1, ident_2, \dots\}) \,\middle|\, \mathfrak{S} \in S \wedge \mathfrak{S} \text{ is in this specific state} \right\}$$

The additional identifiers $ident_1, ident_2, \dots$ can be used to specify the state on a more detailed level per service (see sync example later in this section).

7.1.1. Maintenance

Maintenance tasks include updates of the service to reflect changed or new information needs as well as database optimizations or relocating the metric service to a new host. When maintaining a metric service, the access to it should be prohibited until the service is in normal operational parameters again. For example metric customers who access informations from a maintained metric kernel could see wrong our outdated data.

Let S be the set of all metric services in an EMI. In general we define the maintenance state as:

$$State_{Maintenance} = \left\{ (\mathfrak{S}, \{\}) \,\middle|\, \mathfrak{S} \in S \wedge \mathfrak{S} \text{ is beeing maintained} \right\}$$

Maintenance can be subdivided into planned and unplanned maintenance. The tasks mentioned before are considered planed maintenance. Planed maintenance is typically performed at a time with little access to the system for example the weekend. Unplanned maintenance, however, is required if a (crucial) error is found in the services and maintenance needs to be performed immediately.

7.1.2. Sync with Data Provider

The out-of-sync state represents a specific metric service scenario in which the data that entered the metric service is not the complete current data of all of its data providers. This typically happens if a data provider contains inconsistent data which is not adapted (rejected) from a data adapter or if a data adapter is offline or defective. In such a case the data adapter and all metric kernel that require this particular data are *out-of-sync*. A metric kernel will typically also be out-of-sync if it needs to be (re) set up in a particular EMI.

In general the sync-state is defined for every entity of measurement of the data provider to provide the stakeholders with a finer error indicator. A metric service can, hence, be in-sync for an entity of measurement eom_1 and out-of-sync for another entity of measurement eom_2. We formally include this in the sync state by adding an entity of measurement dimension to the states. Hence, entity of measurement specific states contain tuples of metric services and entities of measurement.

Let $\mathfrak{S}$ be a metric service and eom_1 and eom_2 two entities of measurement like above. The in sync state hence only contains the tuple of the metric service and eom_1 because the service is out-of-sync for eom_2.

$$State_{In-Sync} = \left\{ (\mathfrak{S}, \{eom_1\}) \right\}$$

Let S be the set of all metric services in an EMI and EOM the set of all entities of measurement in all data providers adapted by the EMI. We then define the sync state as:

$$State_{In-Sync} = \left\{ (\mathfrak{S}, \{eom\}) \,\middle|\, \mathfrak{S} \in S \,\wedge\, eom \in EOM \,\wedge\, \mathfrak{S} \text{ is in-sync for } eom \right\}$$

We formally define the ripple effect of the sync state via the feed relation between the metrics. Let $\mathfrak{M}_1$ and $\mathfrak{M}_2$ be two metrics with $\mathfrak{M}_1 \overrightarrow{\approx} \mathfrak{M}_2$ (see section 2.3.8), $\mathfrak{S}_1$ and $\mathfrak{S}_2$ be two EMI services that implement these metrics, and EOM be the set of all entities of measurements. Sync-rippling is then defined as:

$$\forall\, eom \in EOM : (\mathfrak{S}_1, \{eom\}) \notin State_{In-Sync} \Rightarrow (\mathfrak{S}_2, \{eom\}) \notin State_{In-Sync}$$

This already concludes our brief discussion on a formalism for service states. The following section extends our formalism from the foundations to a formal basis for our MeDIC reference architecture.

7.2. Formal Basis of the Technical Reference Architecture

This sections extends our formalism for the conceptual measurement data flow in our metric systems from section 2.3 according to the mapping presented in section 5.1.2. It provides a solid formal approach for the data flow and data processing in an EMI. Using this formal approach we are able to investigate and prove calculation termination of an EMI and correctness of the implementation of the metric kernels. From these investigations we motivate some of the (technical) requirements for Metric Kernels and Data Adapters. Furthermore, the formalism indicates further problem scenarios inside an EMI that an operator or a metric expert should be aware of. Therefore some of the sub sections define additional performance indicators for the monitoring system as well as additional logging information that should be feed to the logging system.

The following sections extend our concepts from section 2.3.4 and section 2.2.3 by implementing the concepts with dedicated parts from the reference architecture. The core responsibility of an EMI is the transport and processing of different types of data for the sake of calculating and presenting metric values. For this we choose a message-based data integration on the EMDB by the means of measurement messages (see section 5.2). Therefore, we first need to extend the formal approaches for measurements from section 2.3.4 in order to reflect measurement messages. We then describe the formal approach to Data Adapters in section 7.2.3 before addressing the Metric Kernels in section 7.2.4. The metric kernel section also describes the Data Storage, Indicator Access APIs and Measurement Result depicted in figure 5.3. We then use the concepts of measurement data compatibility and measurement data matching as a basis for the investigations on the overall data processing in an EMI in section 7.2.5. We close the section with a short summary in section 7.2.6.

We do not include visualization aspects (Monitors) in our formal approach because they are quit hard to grasp formally. We also do not see any benefit in adding this since Monitors *just* visualize the data provided by the indicator access APIs of the metric kernels. They do not (should not) include any calculation. Hence, they can be ignored when investigating termination and reach of an EMI.

7.2.1. Preface

The messages on the EMDB and their reception and sending has a timing aspect. Therefore, we need to be able to formalize timing as well. In particular we need to formalize that a function f_1 is executed before another function f_2 without f_1 providing an input to f_2 [1].

For this formalization we define the ||-operator between two function f_1 and f_2. Like required $f_1 \,||\, f_2$ formalizes that f_1 is executed before f_2. The result of the execution is the result from f_2.

[1] This would simply be $f_2(f_1())$

7.2.2. Measurement Messages

The formalism from section 2.3 is based on measurement data and measurements. In an EMI, however, the core services exchange Measurement Messages. Therefore we need to extend our definitions from section 2.3.4. We formalize measurements in the reference architecture by encapsulating them in measurement messages $\mathtt{m}$. These measurement messages are exchanged over the enterprise measurement data bus (EMDB). In addition to measurements these contain a time stamp ts which indicates the data change time that lead to the measurement. Measurement messages are, hence, defined as:

$$\mathtt{m} = (d, ts, M_{id}, eom) \tag{7.1}$$

Measurement Message Equivalence

Measurement messages are considered equivalent ($\equiv$) if they identify the same measurement, the same entity of measurement, and the same time stamp even if the measurement data is different. Let $\mathtt{m}_1 = (d_1, ts_1, M_{id_1}, eom_1)$ and $\mathtt{m}_2 = (d_2, ts_2, M_{id_2}, eom_2)$ be two measurement messages:

$$ts_1 = ts_2 \wedge M_{id_1} = M_{id_2} \wedge eom_1 = eom_2 \quad \Leftrightarrow \quad \mathtt{m}_1 \equiv \mathtt{m}_2 \tag{7.2}$$

Latency

It is important to differentiate between the data change time and the measurement time because these two will be different! The data change time is the time when the change occurred in the data provider that lead to data d. The measurement time, however, is the time the data was adapted by a data adapter. The difference between the two times is called the *latency* of the measurement message. From this the average latency of a data adapter can be calculated.

Let $\mathtt{m} = (d, ts, M_{id}, eom)$ be a measurement message and let t be the time of the sending of the measurement message by the data adapter. As defined above, the time stamp in the measurement message ts corresponds to the time of the data change in the data provider. The latency L of this measurement message is therefore calculated as the time difference between the data change and the sending of the measurement message:

$$L = t - ts$$

The average latency for a data adapter L_{avg} can be calculated by using the latencies of its produced measurement messages. Let $L_1, \ldots, L_n$ be the latencies for all n measurement messages produced by the data adapter. The average latency (mean latency) of the data adapter can be calculated as:

$$L_{avg} \quad = \quad \frac{1}{n} \sum_{i=1}^{n} L_i$$

The average latency together with the minimum and maximum latency as well as their distribution provides valuable information about the performance of a data adapter for operators. A good indicator for a problem in a data adapter or the communication between a data adapter and a data provider is a sudden increase of the latency. Taken to the extreme: the latencies can be the basis for a service level agreement for the data adaption of an EMI.

Updates

Let $\mathtt{m}_1$ and $\mathtt{m}_2$ be two measurement messages like above. If $eom_1 = eom_2$, $M_{id_1} = M_{id_2}$, $ts_2 \geq ts_1$ ($\mathtt{m}_1$ is older then $\mathtt{m}_2$) and $d_1 \neq d_2$ then $\mathtt{m}_2$ is called an *update* of $\mathtt{m}_1$. This can occur if inconsistent data is corrected in the data provider. For example a spreadsheet with missing data in a row or wrong type of data.

Furthermore, even if $\mathtt{m}_1 \equiv \mathtt{m}_2$ (only the data is different) and $\mathtt{m}_2$ is received after $\mathtt{m}_1$ then $\mathtt{m}_2$ is called a *hard update* of $\mathtt{m}_1$. Hard updates can happen during certain maintenance tasks or during the correction of inconsistent data similar to above. However, they should not occur often or unanticipated. Therefore, the number of hard updates should be monitored and the message that triggers the hard update and the old data should be logged.

Reduction to Measurement

In later sections we need to be able to access the measurement in a measurement message (get rid of the time stamp). Therefore, we define a reduction function that is applicable on a measurement message as follows:

$$\begin{aligned} \mathtt{m}|_{\mathcal{M}} &: \mathtt{m} \rightarrow \mathcal{M} \\ (d, ts, M_{id}, eom)|_{\mathcal{M}} &= (d, M_{id}, eom) \end{aligned}$$

Using this reduction function we define the reduction function for sets of measurement messages as follows:

$$\begin{aligned} \mathtt{M}|_{\mathbb{M}} &: \mathtt{M} \rightarrow \mathbb{M} \\ \{\mathtt{m}_1, \ldots, \mathtt{m}_2\}|_{\mathbb{M}} &= \{\mathtt{m}_1|_{\mathcal{M}}, \ldots, \mathtt{m}_2|_{\mathcal{M}}\} \end{aligned}$$

7.2.3. Data Adapter

A data adapter $\mathfrak{A}$ is responsible for adapting the data from a set[2] of data providers. As defined above in our concept to implementation mapping in section 5.1.2, the data adaption of the data adapter implements a number of measurement functions $f_1, \ldots, f_m$.

[2]Typically each data adapter is just responsible for adapting the data from one data provider. In some cases, however, a data adapter is used for the adaption of several data providers. For example similar data providers on different servers or different data providers with similar data.

The measurement messages produced by the data adapter are based on the adaption of the raw data in a data provider. Therefore, we first need to define the raw data and the corresponding calculation of measurement data from it using the measurement functions.

Let $Raw = \{raw_1, \ldots, raw_t\}$ be the raw data from the data providers feeding the data adapter and $f_1, \ldots, f_m$ be the measurement functions implemented by the data adapter. Each measurement function f_i is able to process specific raw data $Raw_i = \{raw_{k_1}, \ldots, raw_{k_h}\}$ for $k_1, \ldots, k_h \in \overline{t}$. For simplicity reasons, we define that a measurement function is only able to operate on one raw data entry at a time[3]. The output of the calculation of the measurement function on the raw data is therefore the measurement data $d = f_i(raw)$ for all $raw \in Raw_i$.

The measurement messages produced by the data adapter also requires a time stamp and an entity of measurement besides the measurement data. The time stamp and the entity of measurement for a raw data are calculated using specific time calculation and eom calculation functions in the data adapter:

$$\begin{aligned} time_{\mathfrak{A}} &: raw \rightarrow Timestamp \\ eom_{\mathfrak{A}} &: raw \rightarrow eom \end{aligned}$$

We can therefore define the production function of the data adapter as follows:

$$\begin{aligned} produce_{\mathfrak{A}} &: raw \rightarrow \mathtt{m} \\ produce_{\mathfrak{A}}(raw) &= (\, f_i(raw),\, time_{\mathfrak{A}}(raw),\, M_{id_i},\, eom_{\mathfrak{A}}(raw)\,) \end{aligned}$$

Hence, the product of the data adapter is the set of all measurement messages $\mathtt{M}$ resulting from the adaption of the raw data from the data provider. Using the production function we can calculate the product of the data adapter as:

$$\mathtt{M}_{\mathfrak{A}} = \left\{ \; produce_{\mathfrak{A}}(raw) \;\middle|\; \forall\, raw \in Raw_i \;\wedge\; \forall\, i \in \overline{m} \; \right\}$$

7.2.4. Metric Kernel

A metric kernel $\mathfrak{K}$ implements a number of calculation function $f_1, \ldots, f_n$. Hence, they consume measurement messages, provide indicators, and produce new measurement messages based on our formalism in section 2.3.7. In the following sub section we will, therefore, adapt these concepts to measurement messages and the idea behind metric kernels.

[3]This does also makes sense from a conceptual point of view because a measurement function that would require multiple raw data would already perform calculations. Calculation, however, should be performed in metric kernels.

Measurement Message Consumer

In section 2.3.6 we provided a higher order function for the type-based generation of guard functions of measurement consumers. However, as described above, a metric kernel implements a number of calculation functions. Hence, the guard function of the metric kernel must be specified in a way that it accepts all measurement messages that are accepted by one of the guards of the measurement consumers of the derived metrics of the calculation functions (see figure 2.6 for clarification). From now on, for simplicity reasons, we simply abbreviate this by using the concepts of the measurement consumer on the calculation function because their relation is one-to-one anyways.

Let $\mathbb{T}_i$ be the set of accepted types that generate the guard functions for the calculation function f_i. Then, the accepted type set $\mathbb{T}_{\mathfrak{K}}$ that generates the guard function of a metric kernel $\mathfrak{K}$ is calculated as:

$$\mathbb{T}_{\mathfrak{K}} = \bigcup_{i=1}^{n} \mathbb{T}_i \tag{7.3}$$

Note that some of these types may be compatible or equal.

In an actual EMI there needs to be a provider for each of these types. Therefore, a good consistency check for the design of an EMI is to verify that every type set $\mathbb{T}_{\mathfrak{K}}$ for all metric kernels $\mathfrak{K}$ of an EMI can be satisfied by the products of a data adapter or metric kernel.

Handling Measurement Messages

The metric kernel receives the data asynchronously by the means of measurement messages. It, hence, needs to store the data from the measurement messages to enable access to the calculation results (synchronously) at any time. Typically the storage function and the data stored in the metric kernel are defined in a way which eases the calculation of the calculation function. We define the data stored in the metric kernel as $store = \{d'_1, \ldots, d'_l\}$. The data entries in the store are transformed measurement data from the measurement messages.

The data storage of the metric kernel is filled by the storage function *persist* of the metric kernel. This function is defined as:

$$persist : store \times \mathtt{m} \rightarrow store$$

It takes a (compatible) measurement message $\mathtt{m}$ and the current data storage as input, extracts the data d from the measurement message, and transforms and stores it for the metric kernel as new *store*. For simplicity reasons we typically omit the *store* parameter from the storage function and treat the changing of the data store as a side effect of the storage function. This simplifies the following definitions. To revoke this, the data storage parameter and output can be added to all the function that require the storage function if so desired.

Furthermore, we define the extended version of the storage function that works on a set of measurement messages. Let $\mathtt{M} = \{\mathtt{m}_1, \ldots, \mathtt{m}_n\}$ be a set of measurement messages.

We then define the extended storage function as:

$$\begin{aligned} persist &: store \times \mathtt{M} \rightarrow store \\ persist(\mathtt{M}) &= persist(\mathtt{m}_1) \,||\, \ldots \,||\, persist(\mathtt{m}_n) \end{aligned}$$

Due to (possible) changes and alterations to the format of the data the calculation functions are also altered to f'_k. These functions need to provide equivalent results to the original calculation functions. The following paragraph provides the tools to prove this.

To prove the equivalence we need to prove that the calculation result of the tailored calculation function on the data storage is equal to the result of the calculations on unaltered data. Therefore, let $f_1, \ldots, f_n$ be the calculation functions implemented by the metric kernel, $\mathtt{M}$ be a set of measurement messages that is accepted by the guard function of the metric kernel, *store* be the data stored in the metric kernel and $conf_i$ be variability configurations that are compatible with the variability of their function f_i. The metric kernel provides correct indicators and derived measurements (is correctly implemented) iff:

$$\forall\, \mathtt{m} \in \mathtt{M},\, \forall\, i \in \overline{n} : \quad persist(\mathtt{m}) \,||\, f'_i(store, conf_i) \;=\; f_i(\mathtt{m}|_{\mathcal{M}}, conf_i) \tag{7.4}$$

Stability of the Metric Kernel

If the metric kernel breaks down it needs to be set up again. During this setup, due to possible missed measurement messages, it may happen that the metric kernel receives measurements messages that it already processed before[4]. Receiving the same measurement message multiple times, however, must not effect the result of the calculation because it does not contain new data. If the output of the metric kernel does not change when it receives a measurement message multiple times then we call it *stable*.

Formally defined: a metric kernel is stable iff for all measurement messages $\mathtt{m}$, all possible variability configurations $conf$, and all its functions f' the following equation holds:

$$persist(\mathtt{m}) \,||\, persist(\mathtt{m}) \,||\, f'(store, conf) \;=\; persist(\mathtt{m}) \,||\, f'(store, conf) \tag{7.5}$$

Indicator Access

A metric kernel provides a number of indicators for the visualizations, as described in our technical reference architecture above. The indicators are accessed via specific indicator access APIs. Each indicator access API provides a specific data type `dataType` to the visualizations. The indicator access APIs do not calculate any metric but only provide a view or transformation on the results of the calculations from the calculation functions.

[4] Receiving a message multiple times may also happen during regular operation of an EMI based on the messaging infrastructure used to implement the EMDB.

For an indicator access API $\mathcal{I}$ we define the view as a function $\texttt{view}_{\mathcal{I}}$ that consumes the output of a calculation function and provides an indicator according to the data type of the API:

$$\texttt{view}_{\mathcal{I}} : d \rightarrow \texttt{dataType}$$

Using the previews sections we can define the output of an indicator access API on the transformed calculation function f' using the variability configuration $conf$ as:

$$\texttt{view}_{\mathcal{I}}(f'(store, conf)) \tag{7.6}$$

In the technical reference architecture we fully define the access on the indicator access APIs. These require a metric identifier to specify the calculation function. Furthermore, they also require an entity of measurement to filter the output accordingly. However, for our remaining formalism these details are not important[5]. Hence, we only provide the definition from above and leaf the full definition to future work.

Measurement Production

The calculation of derived measurement in a metric kernel follows our formalism for measurement producer in section 2.3.7. The metric kernels provide derived measurements for a fixed set of variability configurations on the calculation functions. However, the metric kernels use altered calculation functions because of their internal storage. Furthermore, the input of the producer needs to be changed from measurements to the data store and its output to measurement messages. Hence, we need to alter the definition for the derived producer.

Let P be a measurement producer of a derived metric implemented in the metric kernel $\mathfrak{K}$, E_P be the calculation function for the entities of measurement, M_{id} be the metric identifier for this product, EOM be the set of entities of measurement in the data store $store$ of the metric kernel, $now()$ be the function that returns the current time (for the time stamp of the measurement message), f' be the altered calculation function of the calculation function f of the derived metric of the measurement producer, and $conf$ be the variability configuration associated with the measurement producer. The production function of a measurement producer of the metric kernel is then defined as:

$$\begin{aligned} produce_P &: store \rightarrow \texttt{m} \\ produce_P(store) &= \Big(f'(store, conf),\, now(),\, M_{id},\, E_P(EOM) \Big) \end{aligned} \tag{7.7}$$

The production function of the metric kernel $\mathfrak{K}$ (similar to section 2.3.7) simply combines the output of the production function for all measurement producers of all derived metrics implemented in the metric kernel:

$$\begin{aligned} produce_{\mathfrak{K}} &: store \rightarrow \texttt{M}: \\ produce_{\mathfrak{K}}(store) &= \Big\{ produce_{P_1}(store),\, \ldots,\, produce_{P_h}(store) \Big\} \end{aligned}$$

[5] Mostly due to the fact that we do not provide a formalism for the visualization frontends.

Calculation Termination for a Metric Kernel

The output of derived metrics for a metric kernel follows our concept for measurement producers from section 2.3.7. Therefore, it is sufficient to show that the metric kernel does not consume its products in a tailored version of equation 2.4 to show termination. Let $\mathbb{T}_{\mathfrak{K}}$ be the set of accepted types of the metric kernel $\mathfrak{K}$, *store* be the data store of the metric kernel, and $produce_{\mathfrak{K}}$ be the production function of the metric kernel. If the following equation holds then the calculation of an EMI with just this metric kernel will terminate.

$$produce_{\mathfrak{K}}(store)|_{\mathbb{M}} \not\models \mathbb{T}_{\mathfrak{K}} \tag{7.8}$$

7.2.5. Data Processing in an EMI

We can use the definitions above to define the outer data processing hull of a complete EMI; its *reach*. This hull represents the calculation output of a combination of several metric kernels and data adapters feed by multiple data providers. Before defining the outer data processing hull, however, we require some additional abstractions to ease its definition.

Robust Metric Kernel Production Function

The calculation functions, and hence the production functions above, are only defined on suitable data input that passes the guard function. To ease the definition of the processing hull we require production functions that are also well defined on non-suitable data. The idea is to extend the definition of the functions. We call this extension the *robust* version of the function because they tolerate "wrong" input.

Let $produce_P$ be the production function for a measurement producer P of a metric kernel, $\mathbb{T}$ the set of types that generate the guard function of the calculation function f from the measurement producer, and *persist* be the extended storage function of the metric kernel. We then define the robust production function $\widehat{produce_P}$ as:

$$\begin{aligned} \widehat{produce_P} &: \mathtt{M} \to \mathtt{M} \\ \widehat{produce_P}(store) &= \begin{cases} \{persist(\mathtt{M}) \,||\, produce_P(store)\} & \text{if } \mathtt{M}|_{\mathbb{M}} \models \mathbb{T} \\ \{\} & \text{otherwise} \end{cases} \end{aligned}$$

Note that, contrasting the definition from before, the output of the robust production function is a set of measurements rather than a single measurement. This eases the following definitions.

Using this we can define the robust production function of a metric kernel $\mathfrak{K}$. Let $P_1, \ldots, P_h$ be all measurement producer of derived metrics implemented by the metric kernel, and $\widehat{produce_{P_i}}$ be their robust production function.

We then define the robust production function of the metric kernel $\widehat{produce}_{\mathfrak{K}}$ as:

$$\begin{aligned} \widehat{produce}_{\mathfrak{K}} &: \mathtt{M} \rightarrow \mathtt{M} : \\ \widehat{produce}_{\mathfrak{K}}(\mathtt{M}) &= \bigcup_{i=1}^{h} \widehat{produce}_{P_i}(store) \end{aligned}$$

Note that only the calculation results for suitable measurement messages are calculated because of the way the robust versions of the measurement producers are constructed. However, the calculation is always well defined and can therefore be applied on arbitrary sets of measurement messages without checking their types. Using these robust production functions for metric kernels we are now able to define the outer data processing hull of an EMI in the following section.

Outer Data Processing Hull

Using our definitions from above and with the definition for the product of a data adapter from section 7.2.3 we are now able to calculate the outer data processing hull of an EMI as the set of all measurement messages produces in the EMI from a set of raw data in the data provider.

Let $\mathfrak{A}_1, \ldots, \mathfrak{A}_n$ be all data adapter in an EMI and $\mathtt{M}_{\mathfrak{A}_i}$ be the product of the data adapter (the measurement messages resulting from the adaption of the raw data from the data provider). We can then define the first data processing step in an EMI $\mathtt{M}^0$ as:

$$\mathtt{M}^0 = \bigcup_{i=1}^{n} \mathtt{M}_{\mathfrak{A}_i}$$

Let $\mathfrak{K}_1, \ldots, \mathfrak{K}_m$ be all metric kernel in an EMI and $\widehat{produce}_{\mathfrak{K}_j}$ their robust production functions. We can then recursively define a processing step in an EMI as:

$$\mathtt{M}^{i+1} = \bigcup_{j=1}^{m} \widehat{produce}_{\mathfrak{K}_j}(\mathtt{M}^i) \qquad i \geq 0$$

Using this recursive definition we can now define the outer data processing hull of an EMI. Let $m \in \mathbb{N}$ be the point at which $\mathtt{M}^m = \mathtt{M}^{m+1}$ another processing step does not produce any new messages. We can then define the outer data processing hull of an EMI $\mathtt{M}^*$ as:

$$\mathtt{M}^* = \bigcup_{i=1}^{m} \mathtt{M}^i$$

The outer data processing hull only surely exists if the calculation in the EMI terminates; otherwise m does not necessarily exist[6]. Following equation 7.4 the metric kernel correctly implements the calculation functions in the metric portfolio. Therefore, the calculation

[6] m can also exist in a non terminating EMI. For example if there exists an infinite calculation of the same measurement then the output will always be identical and m exists.

in the EMI terminates if the calculations in the metric portfolio terminate. Thus, it is sufficient to show that equation 2.5 holds for the metric portfolio implemented in the EMI.

The outer data processing hull contains all measurement messages that result from the adaption of the raw data by the data adapter ($\mathtt{M}^0$). Therefore, we refer to $\mathtt{M}^*$ as the *reach* of the raw data in the EMI. Furthermore, we also refer to the first set of measurement messages from the data adapter $\mathtt{M}^0$ as the *seed* of the raw data in the EMI.

7.2.6. Formalism Summary

This concludes our formalism for the MeDIC reference architecture. We started with a formal extension of measurements to measurement messages. These correspond to the EMDB messages in our technical reference architecture. From there we then provided a formalism for the adaption of raw data from data providers by data adapters. The data adaption, application of the measurement function, and production of measurement messages by data adapters is straight forward and we believe the formalism is easy to use.

We then moved to the more complex formalism for metric kernels. These implement the derived metrics and their calculation functions. We showed how to tailer the existing formalism to suite the metric kernel, for example for measurement consumer and measurement producers. Furthermore, we also provided additional foundations for important properties of the metric kernel like stability and correctness.

Our discussion on the data processing inside an EMI then provided us with a specification of the reach of an EMI (its outer data processing hull). This includes all measurement messages that are generated in an EMI based on the raw data in the data provider. This provides a powerful tool to investigate all generated measurements which helps to direct the increments in our process model in the next part.

This very detailed and formal specification of the concepts in an EMI are typically not required. However, the formal specification of an actual EMI may indicate design problems on an EMI before implementation. Especially proving the correctness of the metric kernel and proving the termination of an EMI are valuable tools to find design problems.

With this formalism we have all tools at hand to formally define all the elements in an EMI based on our MeDIC reference architecture. The ticket statistics example in the next sections shows the application of the formalism and its strengths.

7.3. Formalism Example: Ticket Statistics

This section provides an example for the formal definition of a metric kernel and the data processing inside an EMI. It also shows how we can use the requirements from above to prove termination of the EMI and correctness of the transformation of the calculation functions in the metric kernel.

7.3.1. Introduction and Definition of the EMI

Let $\mathfrak{K}$ be a metric kernel that calculates metrics for simple statistics on ticket management systems. The metric kernel receives its data from the adaption of a ticket management system the *Example Issue Tracker*. The system stores the tickets $t_1, \ldots, t_n$ as its raw data. Each ticket contains a number of dedicated attributes with specific values ($t.attribute = value$). For example a ticket t with high priority and low severity would contain: $t.priority = high$, $t.severity = low$. The data from the data provider is adapted via a data adapter $\mathfrak{A}$. This generates measurement messages containing measurement data similar to the example in section 2.3.5. In this example we focus on the metric kernel because all interesting concepts and proofs are defined for them. Before we start with the kernel, however, we first define the conceptual basis, the metrics, in the following sub section.

7.3.2. Metric Definition

Before we can provide an example for the metric kernel and its formal definition we need to define the actual metrics that are implemented by the metric kernel. We like to use two in this example. Each metric simply counts the number of tickets with different attributes. The first metric $\mathfrak{M}_{HighPrioCount}$ simply counts the number of tickets with high priority without any variability. The second metric $\mathfrak{M}_{VarSevCount}$ counts the number of tickets for a given severity class provided in the variability configuration.

We now need to formally define the calculation functions of the metrics. Because the metrics count the tickets we can simply define the calculation functions using sums. However, to ease the definition we first define the following helper function to check matching attribute values. The *matches* function checks if the given attribute on the given ticket is equal to the given value. If so the function returns 1. Otherwise it returns 0. Let t be a ticket, *attribute* the name of an attribute of the ticket, and *value* be a value that the attribute can hold. The matches function is then defined as:

$$matches(t, attribute, value) = \begin{cases} 1 & \text{if } t.attribute = value \\ 0 & \text{otherwise} \end{cases}$$

Using this function we are now able to easily define the two calculation functions. Let $T = \{t_1, \ldots, t_n\}$ be an arbitrary set of tickets. The two calculation functions for the two

corresponding metrics are then defined as:

$$f_{HighPrioCount}(T, conf) = \sum_{i=1}^{n} matches(t_i, priority, high)$$
$$f_{VarSevCount}(T, conf) = \sum_{i=1}^{n} matches(t_i, severity, conf.vp_{severity})$$

In order to provide a solid formal definition of these two metrics we also need to formally define their variability models and variation points (see section 2.2.4). The first metric does not provide any variability. Therefore its variability model is empty:

$$VM_{HighPrioCount} = \{\}$$

Contrasting the first one, the second metric should be variable. We already saw the application of the variability configuration in the function definition above. The variability model of the second metric provides a closed variability point for the severity. The variability point contains the values: *high*, *medium*, and *low*, which mirrors the different severity states in the ticket management system. Hence, the variability model and variability point for the metric are defined as:

$$VM_{VarSevCount} = \{vp_{severity}\}$$
$$vp_{severity} = \{high, medium, low\}$$

The calculation functions, obviously, require a specific type of data (tickets). Therefore we also need to define the set of accepted types that define the guard function for each of the metrics (and their calculation functions). Both metrics require tickets which, we assume, are identified using a *ticketId* attribute. Furthermore, the metric $\mathfrak{M}_{HighPrioCount}$ requires a data field for *priority* and the metric $\mathfrak{M}_{VarSevCount}$ requires a data field for *severity*. Using our definition for data types from section 2.3.4 we define the following measurement data types:

$$T_{TicketWithPriority} = \{ticketId, priority\}$$
$$T_{TicketWithSeverity} = \{ticketId, severity\}$$

The metrics, however, are defined on measurements not on measurement data. We therefore extend the measurement data types from above to measurement types. These measurement types are then used to generate the guard functions for the metrics. Following our namespace concept, we define `ticket.data` as the root metric identifier for the measurements. We can distinguish between the two types using the specific name spaces `ticket.data.withPriority` and `ticket.data.withSeverity`.

$$\mathcal{T}_{TicketWithPriority} = (\{ticketId, priority\}, \texttt{ticket.data.withPriority})$$
$$\mathcal{T}_{TicketWithSeverity} = (\{ticketId, severity\}, \texttt{ticket.data.withSeverity})$$

Most of the definitions on these types require type sets. We therefore wrap the two types in sets and get:

$$\mathbb{T}_{TicketWithPriority} = \{\, \mathcal{T}_{TicketWithPriority} \,\}$$
$$\mathbb{T}_{TicketWithSeverity} = \{\, \mathcal{T}_{TicketWithSeverity} \,\}$$

Furthermore, the metrics should each provide one measurement producer. These provide measurements for other metrics in the metric portfolio. The first metric $\mathfrak{M}_{HighPrioCount}$, without variability, should simply provide its calculation results as new measurement. We follow the definition of the measurement producer from section 2.3.7. Therefore, for our first measurement producer $P_{HighPrioCount}$ we need to define a new metric identifier for the measurement, a method $E_{HighPrioCount}(EOM)$ to extract the entity of measurement, and a variability configuration $conf_{HighPrioCount}$. We define the metric identifier for the measurement as: `ticket.count.HighPriority` according to our namespace concept. The entity of measurement function ignores the given EOM set and simply returns the name of the ticket management system because the high priority count is defined on all tickets of the system: $E_{HighPrioCount}(EOM) =$ `ExampleIssueTracker`. The metric has an empty variability model. Therefore the variability configuration must be empty as well: $conf_{HighPrioCount} = \{\}$. Following equation 2.2 we can define the measurement producer for the first metric as:

$$produce_{HighPrioCount}(\mathbb{M}) = \Big(f_{HighPrioCount}(\mathbb{M}, \{\}), \quad \texttt{ticket.count.HighPriority}, \quad \texttt{ExampleIssueTracker}, \Big) \tag{7.9}$$

The second metric $\mathfrak{M}_{VarSevCount}$ should similarly provide the count of the high severity tickets. The definition for the measurement producer $P_{HighSevCount}$ of the metric follows the definition from above. However, we must provide a suitable variability configuration to the calculation function to get the count of the high severity tickets. Therefore, let $conf_{HighSeverity}$ be a variability configuration with $conf_{HighSeverity}.vp_{severity} = high$ a variability configuration for the variability model $VM_{VarSevCount}$ of the metric $\mathfrak{M}_{VarSevCount}$. Using the arguments and definition from above we define the measurement producer for the second metric as:

$$produce_{HighSevCount}(\mathbb{M}) = \Big(f_{VarSevCount}(\mathbb{M}, conf_{HighSeverity}), \quad \texttt{ticket.count.HighSeverity}, \quad \texttt{ExampleIssueTracker}, \Big) \tag{7.10}$$

The definition of the metrics is, therefore, complete and we can define the metric kernel

and all its details starting with its measurement consumer in the following sub section.

7.3.3. Metric Kernel: Measurement Consumer

We like to define a metric kernel $\mathfrak{K}$ that implements the two metrics $\mathfrak{M}_{HighPrioCount}$ and $\mathfrak{M}_{VarSevCount}$ defined in the previous sections. Following our formalism for metric kernels we first need to define the measurement consumer. Therefore, we need to define the accepted type set of the metric kernel. Following equation 7.3 the types accepted by the metric kernel are simply the union of the accepted types of the metrics implemented by the metric kernel. Therefore we can define the accepted type set of the metric kernel $\mathbb{T}_{\mathfrak{M}}$ as:

$$\begin{aligned}\mathbb{T}_{\mathfrak{K}} &= \mathbb{T}_{TicketWithPriority} \cup \mathbb{T}_{TicketWithSeverity} \\ &= \Big\{ \, (\, \{ticketId, priority\}, \texttt{ticket.data.withPriority}\,), \\ &\qquad (\, \{ticketId, severity\}, \texttt{ticket.data.withSeverity}\,) \, \Big\}\end{aligned}$$

Using our concepts for compatibility on measurements (see section 2.3.5) we can define an overall measurement type for ticket data as:

$$\mathcal{T}_{Ticket} = (\, \{ticketId, priority, severity\}, \texttt{ticket.data}\,)$$

Following our compatibility definition this type is compatible to the two other types:

$$\mathcal{T}_{Ticket} \prec \mathcal{T}_{TicketWithPriority} \qquad\qquad \mathcal{T}_{Ticket} \prec \mathcal{T}_{TicketWithSeverity}$$

We can therefore simplify the accepted type set of the metric kernel to the wrapped type $\mathcal{T}_{Ticket}$ which provides all required attributes:

$$\mathbb{T}_{\mathfrak{K}} = \Big\{ \, (\, \{ticketId, priority, severity\}, \texttt{ticket.data}\,) \, \Big\}$$

After this definition of the measurement consumer of the metric kernel we define the internal data storage of the metric kernel in the next section.

7.3.4. Metric Kernel: Data Storage

In section 7.2.4 we already mentioned that the data storage in the metric kernel will typically differ from the data in the measurement messages to ease the calculation. The easiest naive way for the calculation in this example would be to pre-calculate and store the calculation results (increase or decrease the values on message receive). Sadly, such a metric kernel would not be stable. If the kernel would receive the same ticket twice then it would also increase the count twice. Hence, equation 7.5 would not hold. Therefore, we will build the storage based on the ticket ids to allow the detection of messages that were already received.

Our goal with the store, however, should be to optimize the calculation. Therefore, the metric kernel stores not one but four sets of ticket ids in its data store. One set for the ticket ids from the tickets with high priority and one for the tickets ids for each severity type (see variability point). The data store is therefore defined as:

$$store = (S_{HighPrio}, S_{HighSev}, S_{MedSev}, S_{LowSev})$$

The storage function of the metric kernel *persist* needs to fill these sets for each measurement message accordingly. For this example we again define a helper store function to ease the definition of the actual storage function. The idea of this helper function is to perform the storage for one data set that is provided as a parameter. The helper function then returns the altered data set. The helper function checks if a given ticket has a given value for a given attribute. If it has then the helper function adds the id of the ticket to the data set. If it does not have the given value then the helper function performs a set minus operation with the ticket id. If the id is in the set this will take it out. If the id is not in the set this does nothing. With this we reflect the fact that the ticket will change some values throughout its live. Hence, if through an update the ticket does not belong to the set anymore it needs to be taken out.

Let t be a ticket, *dataSet* a data set of the internal store, *attribute* the name of an attribute of the ticket, and *value* a value that fits the attribute of the ticket. We then define the helper function $persist'(t, dataSet, attribute, value)$ for a single internal data set as:

$$persist'(t, dataSet, attribute, value) = \begin{cases} dataSet = dataSet \cup \{t.ticketId\} & \text{if } t.attribute = value \\ dataSet = dataSet \setminus \{t.ticketId\} & \text{otherwise} \end{cases}$$

Using this helper function we are now able to define the actual storage function of the metric kernel $persist(\mathtt{m})$. Let $\mathtt{m} = (d, ts, \mathtt{ticket.data}, eom)$ be a measurement message. We then define the storage function as:

$$persist(\mathtt{m}) = \begin{cases} S_{HighPrio} = persist'(\mathtt{m}.d, S_{HighPrio}, priority, high) \\ S_{HighSev} = persist'(\mathtt{m}.d, S_{HighSev}, severity, high) \\ S_{MedSev} = persist'(\mathtt{m}.d, S_{MedSev}, severity, medium) \\ S_{LowSev} = persist'(\mathtt{m}.d, S_{LowSev}, severity, low) \end{cases}$$

The format of the stored data differs from the data in the data provider. Hence, the calculation functions need to be changed, as mentioned above. The sets already store the correct ticket ids. Hence, the function simply needs to count these ids to return the

number of tickets with the corresponding condition.

$$
\begin{aligned}
f'_{HighPrioCount}(store, conf) &= |S_{HighPrio}| \\
f'_{VarSevCount}(store, conf) &= \begin{cases} |S_{HighSev}| & \text{if } conf.vp_{severity} = high \\ |S_{MedSev}| & \text{if } conf.vp_{severity} = medium \\ |S_{LowSev}| & \text{if } conf.vp_{severity} = low \\ -1 & \text{otherwise} \end{cases}
\end{aligned}
$$

We now defined the storage function but we still need to prove that the altered calculation function and the storage function actually implement the requested calculation function. We perform this proof in the next subsection.

7.3.5. Correctness Proof of the Storage Function

To prove the correctness of the storage function we need to prove equation 7.4 for all calculation functions and all measurement data. To save space and because these proves are not too complicated the following section only contains a prove for the first calculation function $f_{HighPrioCount}$ and its transformed function in the metric kernel $f'_{HighPrioCount}$. As a reminder, to prove the correctness we need to prove that the output of the transformed function after the storage function is the same as the output of the unaltered calculation function. However, the basis of the calculation function are tickets but the basis for the storage function are measurement messages. We therefore define a extraction function for the measurement data from a measurement message:

$$
\begin{aligned}
extract &: \mathtt{m} \rightarrow ticket \\
extract(\mathtt{m}) &= m.d
\end{aligned}
$$

However, the calculation functions are defined on sets of tickets. Using the extraction function from above we define another helper function *strip* that "strips" a set of measurement messages and reduces it to a set of ticket data.

Let $T = \{t_1, \ldots, t_n\}$ be a set of tickets and $\mathtt{M} = \{\mathtt{m}_1, \ldots, \mathtt{m}_m\}$ be a set of measurement messages. We then define the strip function as:

$$
\begin{aligned}
strip &: \mathtt{M} \rightarrow T \\
strip(\mathtt{M}) &= \bigcup_{i=1}^{m} extract(\mathtt{m}_i)
\end{aligned}
$$

For a compact definition of equation 7.4 we also define an extension to the storage function that works on a set of measurement messages. This helps us to get rid of the all-quantifier in equation 7.4. The extended storage function simply applies the storage function for each measurement message in the set.

Let M be a set of measurement messages like above. We then define the extended storage function as:

$$\begin{aligned} PERSIST &: \mathtt{M} \rightarrow store \\ PERSIST(\mathtt{M}) &= persist(\mathtt{m}_1) \,||\, \ldots \,||\, persist(\mathtt{m}_m) \end{aligned}$$

Using the strip function and the extended storage function we are now able to fill equation 7.4 with our actual functions. The variability configuration *conf* in the equation can be set to the empty set because the function has an empty variability model (has no variability). Let M be a set of measurement messages like above. The implementation of the calculation function in the metric kernel is correct if we prove the following equation:

$$PERSIST(\mathtt{M}) \,||\, f'_{HighPrioCount}(store, \{\}) \stackrel{!}{=} f_{HighPrioCount}(strip(\mathtt{M}), \{\}) \tag{7.11}$$

First we fill in the actual definition of the functions from above. Let M be a set of measurement messages like above and $strip(\mathtt{M}) = \{t_1, \ldots, t_n\}$ be the set of tickets that we get when stripping M. The metric kernel implementation is correct iff:

$$PERSIST(\mathtt{M}) \,||\, |S_{HighPrio}| \stackrel{!}{=} \sum_{i=1}^{n} matches(t_i, priority, high) \tag{7.12}$$

We now define a view on the ticket set that just contains the high priority tickets. Let M be a set of measurement messages like above and $strip(\mathtt{M}) = \{t_1, \ldots, t_n\} = T$ be the set of tickets that we get when stripping M. We then define the high priority view on these tickets as:

$$T|_{High} = \{t \,|\, t \in T \,\wedge\, t.priority = high\}$$

The matches function of the calculation function on the right hand site in equation 7.12 is parameterized in a way that it only returns 1 if the priority of the ticket is high. Let $T = \{t_1, \ldots, t_n\}$ be a set of tickets like above and $T|_{High} = \{t'_1, \ldots, t'_m\}$ be the high priority view on this set. From the definition of the matches function follows that:

$$\sum_{i=1}^{n} matches(t_i, priority, high) = \sum_{i=1}^{m} matches(t'_i, priority, high)$$

Using the calculation functions (using the same definitions as above) we therefore know that:

$$f_{HighPrioCount}(T, \{\}) = f_{HighPrioCount}(T|_{High}, \{\}) \tag{7.13}$$

The construction of the storage function uses the storage helper function from above. In the actual storage function the helper function for the high priority tickets is parameterized as: $persist'(\mathtt{m}.d, S_{HighPrio}, priority, high)$. If we fill these parameters directly into the

definition of the helper function we get:

$$\ldots \quad = \quad \begin{cases} S_{HighPrio} = S_{HighPrio} \cup \{t.ticketId\} & \text{if } t.priority = high \\ S_{HighPrio} = S_{HighPrio} \setminus \{t.ticketId\} & \text{otherwise} \end{cases}$$

This storage helper function only fills in the high priority tickets from the ticket set. Each ticketId is only contained once in the tickets set because it represents a snapshot of a ticket management system. Therefore, we can also define the high priority data store set using the high priority view on the ticket set from above. Let T be a set of tickets and $T|_{High}$ be the high priority view on this set. We then know that after $PERSIST(\texttt{M})$ the high priority data set just contains the ticket ids from the the high priority view on the tickets:

$$S_{HighPrio} = \{\ t.id \ \mid \ t \in T|_{High}\ \}$$

Using this we can get rid of the storage function in equation 7.12. Let $T|_{High} = \{t'_1, \ldots, t'_m\}$ be the high priority view on the ticket set that resulted from stripping the set of measurement messages like above. Following our discussion from before, the implementation of the calculation function in the metric kernel is correct if the following equation is true:

$$\left| \{t.id \,|\, t \in T|_{High}\} \right| \ \stackrel{!}{=} \ f_{HighPrioCount}(T, \{\})$$

Using equation 7.13 we can further simplify the equation to:

$$\left| \{t.id \,|\, t \in T|_{High}\} \right| \ \stackrel{!}{=} \ f_{HighPrioCount}(T|_{High}, \{\})$$

This can be further simplified by reformatting the sum on the left hand side and filling in the definition for the calculation function on the right hand side:

$$|\ \{\ t'_1.id,\ \ldots,\ t'_m.id\ \}\ | \ \stackrel{!}{=} \ \sum_{i=1}^{m} matches(t'_i, priority, high)$$

The set $T|_{High}$ only contains high priority tickets. We can therefore get rid of the matches function on the right hand side, because it returns 1 for every ticket in the set, and end up with:

$$|\ \{\ t'_1.id,\ \ldots,\ t'_m.id\ \}\ | \ \stackrel{!}{=} \ \sum_{i=1}^{m} 1$$

Both left and right hand side of the equation will result in m. We therefore proved the equivalence in equation 7.11! Hence we proved equation 7.4 for this calculation function of the metric kernel. From this follows the correctness of the implementation of this calculation function in the metric kernel! □

7.3.6. Metric Kernel: Measurement Producer

According to the metric definition from above the metric kernel should provide two measurement producers $produce_{HighPrioCount}$ and $produce_{HighSevCount}$. These producers should send measurement messages for the number of high priority tickets and the number of high severity tickets. According to equation 7.7 and using the definitions for the measurement producers in equation 7.9 and equation 7.10 directly follows the definition for the measurement producers of the metric kernel.

The first measurement producer for the high priority ticket count from equation 7.9 is implemented by the following producer from the metric kernel:

$$\begin{aligned} produce_{HighPrioCount}(store) = \Big(& f'_{HighPrioCount}(store, \{\}), \\ & now(), \\ & \texttt{ticket.count.HighPriority}, \\ & \texttt{ExampleIssueTracker}, \\ & \Big) \end{aligned}$$

The second measurement producer for the high priority ticket count from equation 7.10 is implemented similarly. Using our definitions for the variability configuration $conf_{HighSeverity}$ to count high severity tickets from above the following producer from the metric kernel implements the high severity producer:

$$\begin{aligned} produce_{HighSevCount}(store) = \Big(& f'_{VarSevCount}(store, conf_{HighSeverity}), \\ & now(), \\ & \texttt{ticket.count.HighSeverity}, \\ & \texttt{ExampleIssueTracker}, \\ & \Big) \end{aligned}$$

Therefore the production function of the metric kernel is defined as:

$$produce_{\mathfrak{K}}(store) = \Big\{ produce_{HighPrioCount}(store),\ produce_{HighSevCount}(store) \Big\}$$

Using this production function we can now (conceptually) prove the termination of the calculation of the metric kernel inside the EMI.

7.3.7. Termination Proof of the Kernel and the EMI

To prove termination of this example we need to prove the requirements from equation 2.5 for all calculation functions. The metric kernel described above is the only metric kernel in this EMI. Hence, satisfying equation 7.8 will also satisfy equation 2.5.

To prove equation 7.8 we just need to consider the metric identifiers of the produced measurement messages and the required data type. The two metric identifiers for the produced measurement messages are: `ticket.count.HighPriority` and `ticket.count.HighSeverity`. The only type in the required type set of the metric kernel $\mathbb{T}_{\mathfrak{K}}$ is the type $\mathcal{T}_{Ticket}$. The type identifier of this type is `ticket.data`.

From our definitions for measurement compatibility in section 2.3.5 follows that:

$$\texttt{ticket.count.HighPriority} \not\prec \texttt{ticket.data}$$
$$\texttt{ticket.count.HighSeverity} \not\prec \texttt{ticket.data}$$

They are not compatible because they are longer and they are not prefixes of `ticket.data`. This also makes sense because their measurement messages obviously provide different data.

Following our definition for satisfiability from section 2.3.6 it directly follows that no production functions from the metric kernel satisfies the accepted type set of the kernel:

$$produce_{HighPrioCount}(store)|_{\mathbb{M}} \not\models \mathbb{T}_{\mathfrak{K}}$$
$$produce_{HighSevCount}(store)|_{\mathbb{M}} \not\models \mathbb{T}_{\mathfrak{K}}$$

Hence, the overall production function also does not satisfy the accepted type set of the metric kernel:

$$produce_{\mathfrak{K}}(store)|_{\mathbb{M}} \not\models \mathbb{T}_{\mathfrak{K}}$$

Thus, the guard function of the metric kernel will not accept any of the produced measurement messages. This proves that there is no feedback in the metric kernel and equation 7.8 is shown. As stated above, from this also follows equation 2.5. Therefore, there are not infinite calculation chains in the EMI and the calculation terminates.

□

7.3.8. Example Summary

This section provided a detailed example of the application of our formalism for our MeDIC reference architecture. We showed how to specify metrics, their variability, and measurement producers. From there we implemented these metrics in a metric kernel using our concepts from the formalism. Furthermore, we showed how to utilize these formalism concepts to prove the correctness of the implementation in the metric kernel and the termination of the EMI of the example. Thus, the example provides a valuable source for the formal description of an actual EMI and proofs associated with it.

This concludes the definition of our reference architecture for enterprise measurement infrastructures in our metric systems engineering approach MeDIC. The following part III will define the second key part of MeDIC: our metric systems engineering process model.

Part III.

Metric Systems Engineering

Process Model

8

Process Model Foundations

The previous section provided a thorough and specific discussion of our reference architecture for enterprise measurement infrastructures (EMIs). In there our logical reference architecture already provided some insides into our engineering approach. Continuing this discussion, this part provides additional details to the design of our process model. We include overviews of the phases and detailed information to selected activities. Further information, checklists, and document definitions are included in appendix B.

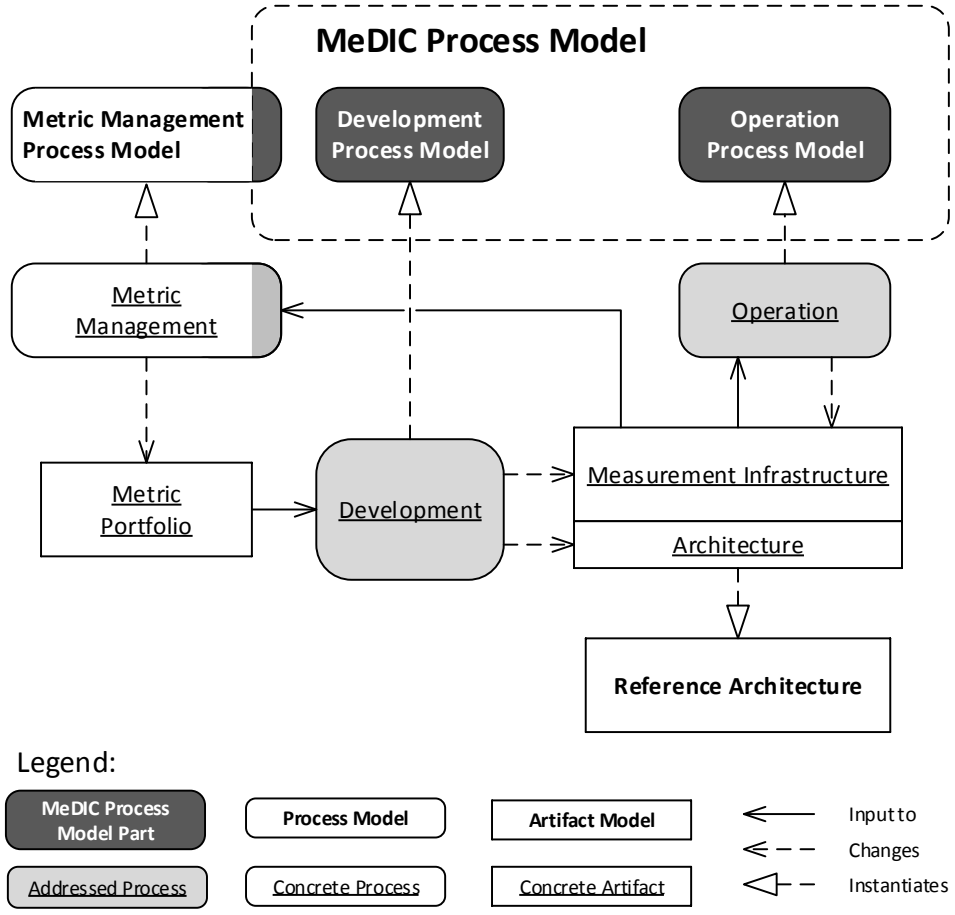


Figure 8.1.: Scope of the metric systems engineering process model

Figure 8.1 provides an overview over the scope of our metric systems engineering process model. We mainly provide process models to define development and operation processes. However, some of the activities in our process model address metric management aspects as well. The main metric management aspect that we address with our process model is finding, evaluating, and modeling metrics in the metric portfolio. We see this also as part of the development because the information needs and metric form the requirements for the actual services in an EMI. In our process model we refer to this part as the conception-phase because it focuses on the concepts (requirements) for the measurement infrastructure.

The core process, that we support with our process model, is the development process. The focus of the process is to take information needs and provide an according measurement infrastructure. Therefore, the process is a classic software development process that was tailored to the specifics of developing an EMI based on our reference architecture. Our iterative and incremental development process model provides activity and artifact blueprints which assist the design, construction, test, and deploy activities. Our description of this part of the process model focuses on the design activities because large parts of the construction, test, and deployment activities are technology specific.

Contrasting a lot of other measurement process models, our process model explicitly addresses the operation of the measurement infrastructure. However, similar to the later development activities, specific activities in the operation phase are also heavily dependent on the implementation technology of the particular EMI. Thus, we will provide details and solutions to typical (technology independent) problems, mainly focusing on service failure, that operations need to face.

The design and elements of the reference architecture obviously influence the design of the process model and vice versa. Hence, a lot of activities in the process model use the technical concepts proposed in the reference architecture. We also use the concepts from our logical reference architecture, for example our concept of *metric applications*, to specify and focus the development increments. Nevertheless, the activities that we define in this process model also influence the services and design of the reference architecture. For example, the message gateway service and message cache service (see section 5.2.4) are required for the test and operation activities defined in the process model.

8.1. Process Environment Assumptions

Similar to other models, our engineering process model is not designed to work in an arbitrary environment. Therefore, we like to discuss a suitable environment for our process model before we start to introduce our process model in the following chapter.

Most importantly, our process model is, in its core, a software engineering process model for metric systems. Therefore, the company applying this process model should have solid software engineering knowledge. This applies to existing processes as well as staff training. Our process model is influenced by modern software development process models. Therefore, experience with iterative and incremental software development should exist to the organization. Furthermore, our process model also covers operation activities. Therefore, the company should already be familiar with operating infrastructures based on loosely coupled services. Particularly, the company should already employ specialized operation staff.

We also assume that at least one trained *metric expert* is working in the company. The metric expert(s) should be familiar with the internal processes at the company because the needs of metric customers often raise from these. Furthermore, they should be familiar with measurement processes and measurement-based improvement methods like CMMI M&A or the ISO 15939 [Tea10, ISO07][1]. We further describe the responsibilities of all the roles further down in section 9.3.

The microservice-based reference architecture enables to freely choose among a large variety of technologies for the different types of services in an EMI. We assume that the architects are able to use this strength accordingly and select suitable technologies based on the actual requirements for a specific service. We also assume that the developers are trained on the technologies and able to implement the designs from the architects. Furthermore, we assume the company provides a specialized EMI development platform to ease development. This platform should contain the adaption for the messaging infrastructure of the EMDB (message base-types, base message senders, and base message receivers) as well as easy to use frameworks for different situations; like adding specific indicators to the monitoring client agent, using the directory client agent, or providing an indicator access API. This development platform can also contain specialized tools and generators to ease certain development tasks.

We further assume the company is mature and can handle its infrastructure accordingly. With this we assume they are familiar with staging and releasing software that contains several services, topic configurations, and databases. All these things may change in one development increment and hence need to be deployed and tested in the different staging environments accordingly. We also assume a pre-production staging environment that can either simulate or provide all data providers for the EMI to run complete system tests. We will briefly go through the stages in section 12.1 and discuss different test phases in the appendix in section B.2.3.

[1] A good start as well as source for guidance and expertise is available at the special measurement and analysis web page from the SEI: `https://www.sei.cmu.edu/measurement/index.cfm`.

9

The Metric System Engineering Process Model

This chapter will provide an overview about our process model. We will describe the different stakeholders, phases, and initial activities before using the model. The description of the detailed activities and artifacts are contained in the next chapters each focusing on a particular phase. In general we try to provide an overview over the activities and design alternatives and reasoning in the the main part of the thesis. The appendix will contain additional information for the actual application of the process model; like checklists, guidelines, and detailed artifact descriptions. We can further provide detailed artifact templates and examples upon request.

The introduction already presented a very generic overview of our view on metric systems engineering in figure 8.1. The very basic idea of our process model are the following steps:

Step 1 Gather (and validate) changed and new information needs from metric customers.

Step 2 Design changes in metric portfolio based on the information needs and use monitor prototypes for validation.

Step 3 Design (and validate) an EMI based on our reference architecture.

Step 4 Develop, test, and stage infrastructure elements based on the design.

Step 5 Deploy elements and operate the EMI.

These steps should be performed in an iterative manner based on increments defined by clusters of information needs. Based on these steps we will present the core of our process model in the following section 9.1. The core will define the process phases. It will also present details on the iterative and incremental approach of the process model. The next section 9.2 will provide a first overview over the core activities in each process phase. Section 9.3 gives detailed descriptions of the needs and responsibilities of the different roles involved in the process model. We will also present their involvement in the process phases similar to the overview picture of the unified process. Section 9.4 will finish this chapter with a description of the activities that should be carried out before going into the first iteration of the process (sometimes referred to as phase 0 or phase -1).

In our discussions and descriptions in the following section we use **`a bold mono spaced font`** to indicate activities of the process model. Furthermore, we use `a normal weight mono spaced font` to indicate artifacts which are produced or consumed in the activities. We do not use a specific font to indicate roles because we just use five roles that are easily identified.

9.1. Process Model Core

We already discussed some important aspects and the foundations for our process model: iterative and incremental development process models, classic measurement processes and best-practices/success factors, and modern software engineering best practices including prototyping, microservice-based designs, and staging/release processes. This section provides an overview over the core of our process model. We first provide a description for the process phases, increments, and iterations. From there we will then present the first (very rough) overview of the process model.

Our process model should aligns well with the ISO 12207 standard for systems and software engineering [ISO08]. From our discussion above we should also include important related success factors; for example success factors for portal applications from Ulrich Remus [Rem07]. Therefore, our process model is rooted in a standard PDCA-cycle, which is often seen as the foundation for innovative engineering based on critical thinking [Lik04]. It is also the basis for many metric-management process model as well as software engineering process models.

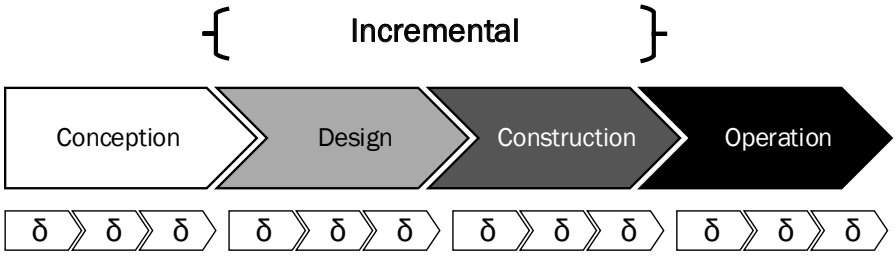


Figure 9.1.: Metric Systems Engineering Process: phases, increments, and iterations

Figure 9.1 provides an overview over the phases of our concept model as well as the iterative and incremental aspects of the process model as Chevron process diagram. The center shows the four phases of our process model: *Conception*, *Design*, *Construction*, and *Operation*. They implement the PDCA cycle in the following roles:

Conception (Plan) The goal of the conception phase is to identify, classify, and weight all information needs of relevant metric customers. These needs form the requirements of the metric system. They are the justification for the metrics in the metric portfolio and consequently the basis for all services and elements in the measurement infrastructure. Our process model assists the identification of information needs with recommendations and discussions on different methods for requirements elicitation. Furthermore, we emphasize the importance of requirements validation after the elicitation. Specifically, we recommend the use of prototypes to validate the solutions to the information needs with metric customers. The conception phase also defines the development increments based on different criteria.

Design (Act) The design phase takes the information needs for this development increment and designs all necessary services and associated tests based on our reference architecture. With *designing* we not only refer to the design of new services but it also includes designing service changes and designing service removal and transition. The result of the design phase is the design document for the specific development increment, which guides the construction.

Construction (Do) The services designed before are implemented, tested, and released in the construction phase. The implementation uses whatever technology may be suitable for the given service (specified in the design document). Testing and release are then executed in typical staging steps with their respective environments. In the later stages (system- and integration-test) we heavily rely on services defined in the reference architecture like the measurement cache, the message gateway, the monitoring service, and the EUrEKA registry to either drive the test, check results, or mock other services.

Operation (Check) This phase, as the name suggests, contains operation activities. These activities define behavior fragments for critical operation tasks like setup of a new service, actions on service failure, or planed service maintenance. Furthermore, this phase contains actions to constantly evaluate the usage of the measurement infrastructure and needs from metric customers to initiate succeeding iterations. Thereby, the metric portfolio and measurement infrastructure will constantly evolve reflecting the current needs of the metric customer without to much overhead of irrelevant metrics.

Pin-pointing a "correct" solution that will satisfy the information needs of the metric customers is hard. Hence, especially when starting the development of a metric system and in the conception phase "errors" are very likely to occur. The process should address this fact in a proactive manner (rather then neglecting it) by planning to do a task several times including appropriate feedbacks in between. Therefore, each of these phases can (and should) be executed in multiple iterations indicated by the δ in figure 9.1.

Multiple iterations in the conception phase, for example, can be used to pin-point the information needs of the metric customers or to include and aggregate information needs from multiple metric customers. In the design phase the architects can use the

iterations to design one service or one information need at a time. Similarly, the developers can work on the implementation of one service at a time. Therefore, they localize the changes which reduces errors. Operations focuses on the evaluation of the satisfaction of information needs of metric customers per iteration as well as continuous operation of the measurement infrastructure which could also be seen as small iterations. Also the overall process should be iterated (see life cycle) because the information needs will change over time. Furthermore, even assuming prototyping and metric customer involvement would be perfect, which it typically is not, it is still hard to get the metric system “right".

As stated in the introduction metric systems may become large (large metric portfolios and complex measurement infrastructures). However, as stated in section 1.5.1 and section 1.5.2 the metric system should not be introduced in a Big-Bang. Therefore, it should be planed, specified, developed, tested, and deployed in multiple (smaller) increments. These increments should be small enough to still be able to manage the given task without too much complexity. Therefore, they should be defined on a coherent set of requirements. We focused our increments either on a coherent set of needs of metric customers or on a set of coherent (technical) services; for example event counting metrics (ticket count, change count, login count, and so on) or metric to support earned value analysis from project management. An increment is then, typically, reflected by a logical metric application (see our logic reference architecture in section 4) throughout the process.

9.2. Process Overview

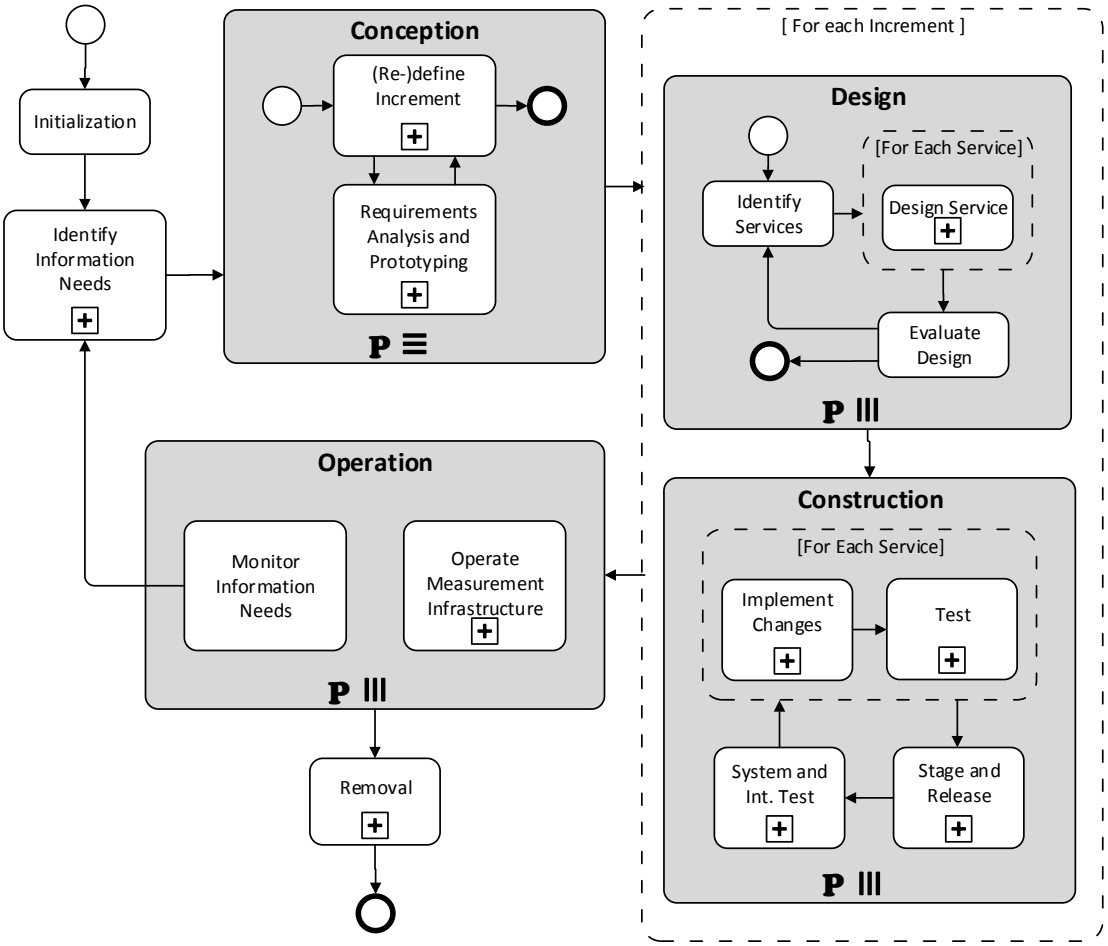


Figure 9.2.: MeDIC engineering process model: phase details and core activities

Figure 9.2 provides an overview of the main activities for each phase in the metric systems development process model as BPMN diagram[1]. Similar to agile software development approaches the most phases are not strictly separated from one another. Most importantly, the operation activities need to be performed contentiously once an increment reaches production. Traditional process phases imply certain semantics. For example they need to define quality gates that need to be satisfied before exiting the phase. In our process model the phases just act as containers for their underlying activities. However, especially the conception phase contains a number of evaluation activities which act similar to quality gates.

After an initial identification of information needs the process enters the conception phase. The main focus of this phase is to find the requirements for and design the

[1]We extended the notation by adding a **P** to the activities that represent our process phases.

increments in which the metric system will be developed or enhanced. We included (and heavy focus on) prototyping in this phase to validate the requirements because it supports the development of uncertain and hard to handle requirements, like it is the case with metric systems. The transition from conception to design may only occur if the evaluation of the prototype is positive and only small issues are found. If the evaluation results in a large number of changes and issues then the conception phase needs to be reiterated including another evaluation step at the end. The detailed activities and artifacts for this phase are described in chapter 10.

The *design* phase focuses on transforming the concepts specified in the conception phase into changes or additions to the measurement infrastructure. In the design phase the first important step is to identify the services that need to be (re) designed in this increment. The design of the service itself is performed iteratively. The metric customers as well as the metric experts should then validate the design before the start of the construction phase. This phase may also include the construction of additional (vertical) prototypes to test certain functionality before the start of the actual construction of the metric services.

The focus of the following construction phase is the development of new metric services or the change of existing services to satisfy the design and concepts from the last phase. Again, we recommend to implement the changes per service (or per aspect) in a separate iterations. Another important aspect in the phase is testing. The phase may only be completed if the tests (system tests and integration tests) are positive after staging the metric services. Furthermore, the finished services need to be evaluated with metric experts and metric customers in a pre-production stage. If major changes are found they need to be addressed in another iteration between design and construction or they may even force a reiteration of the conception phase if several major issues are found. If the evaluation turns out positive, however, the corresponding services of the EMI can be staged to production.

After construction, the process enters the operation phase. The most important activity here is to ensure that the services in the EMI perform within given parameters; to *operate* the EMI. However, after some time, the information needs of the metric customers may have changed (see chapter 1.1.3). It is important to be aware of this fact and constantly evaluate the metric usage as well as gather random samples of information needs of metric customers. This monitoring activity can then spawn a new iteration of the process.

Inevitably, at some point of time a metric system (or parts of it) needs to be removed. Typically, the removal of a metric system and/or its EMI (part) includes the transition to a follow-up metric system and/or different measurement infrastructure.

9.3. Roles

This section will provide an overview of the central roles in our development process model as well as their essential needs. Furthermore, similar to other process models like the open unified process[2], we will also define the central responsibilities for each role. Contrasting popular process models, however, we will not provide detailed lists of all activities, artifact responsibilities, and work products in this thesis because these are hard to read and better accessible via an interactive process documentation[3].

Together with the needs (thinks people in the role want) we also provide a set of dislikes (thinks people in the role do not want) because these are similarly (if not even more) important. The needs and dislikes in the role descriptions are derived from our experience from our field studies.

9.3.1. Metric Customer

Metric customers are, as the name suggests, the customer of the metric system. They want to use the metric system to get answers to their information needs. Unfortunately, the answers to the different information needs are typically stored in many different tools. Furthermore, our experience with many industry partners shows that the information needs of measurement customers often change over time. For example, development tools are replaced by other tools or processes and organization schema are changed. Especially reorganizations lead to new and changed responsibilities of individual measurement customers and roles which inevitably lead to changes in information needs (also see section 1.1 as well as our literature for further discussions and experiences [VLJ13, VL14]). A project manager is a typical example of a measurement customer.

A metric customer...

...wants information needs answered. As described above, the main need of a metric customer is to answer her informations needs using monitors that show indicator data from metrics.

...wants up-to date and correct data. Measurement customers demand correct and up-to-date data because old or incorrect data leads to wrong conclusions and wrong decisions. Hence, an EMI should provide mechanisms that guarantee a fast recognition and processing of relevant events. Additionally, up-to-date data requires robustness and high availability of the EMI.

[2] See `http://epf.eclipse.org/wikis/openup/publish.openup.base/customcategories/role_set_list_199A969B.html?nodeId=e3ed2eb4` for the role descriptions of the open UP.

[3] For example like the typical UP documentations created by the Rational Method Composer. See for example the open unified process documentation at `http://epf.eclipse.org/wikis/openup`

. . . wants to tailor metrics and visualizations. Not all metric customers and not all situations they are in are similar. Hence, they need to tailor the monitors to, more specifically, answer their needs. For example they need to tailor the time frame for a specific monitor from monthly to weekly or daily if they require a higher resolution of the metric.

. . . wants low latency. The time difference between data-change and measurement update should be as low as possible because solving problems should be immediately visible in the visualizations.

. . . does not want technical details. The metric customer does not (and should not!) care about implementation and technical details of the measurement infrastructure. Technical aspects should be hidden as best as possible from the metric customers.

. . . does not want to spend too much time with set-up and specification. Metric customers have important roles in the company and want quick answers from the metric system. Their job is not just to be a metric customer. Therefore the time to set-up and use the metric system and tools in the EMI should be as little as possible.

Responsibilities

The central responsibilities of metric customers in our process model are to . . .

- . . . use the EMI to satisfy some of her information needs.
- . . . provide requirements for the metric system.
- . . . provide feedback concerning the metric system (metrics, tool usability, process integration, . . .).
- . . . evaluate requirements, design, and construction increments.
- . . . help to specify (realistic) system test cases for the services in the measurement infrastructure.

In the actual process model, these responsibilities expand to dedicated activities.

9.3.2. Metric Expert

The metric expert is the central functional roles in our metric systems engineering process model. Metric Experts not only guide metric customers and provide assistance they are also responsible for the maintenance of the metric portfolio and the definition and focus of the development increments. Therefore, metric experts need to be in a manager position at the company. Metric experts, as the name suggests, also need extended knowledge of metrics and visualization theory. Using this knowledge they know what operations are allowed (make sense) on what types of metrics as well as selecting suitable visualizations for the monitors. This knowledge should also be used to optimize the metric portfolio and to design the most suitable monitor to answer a certain need from a metric customer.

The metric expert...

...wants to help the metric customers. Most importantly, metric experts want to help the metric customers. The metric customers typically have a large number of information needs that they need answered in their daily work. However, they not necessarily know how to answer the questions. Therefore, the experts want to find and/or develop new metrics for these unanswered or difficult information needs. Hence, the metric experts can use workshops and interviews to investigate the needs together with the customers and specify according indicators and visualizations to be developed in the EMI.

Furthermore, the experts can provide templates and best-practices for metrics and dashboards. These can then be used by metric customers to answer typical questions. They can also be used for mandatory monitors often used in company reports. The templates and best-practices are also part of the metric portfolio.

...wants to optimize the metric portfolio. The metric experts are responsible for managing the metric portfolio. Therefore, they typically want to optimize it. Optimizing, in this situation, means removing unused metrics, finding improvement potentials for used metrics, and finding blind-spots. Improving existing metrics can refer to optimizing a calculation or optimizing the variability model to ease or enhance configuration of the metric. Optimizing the portfolio requires that the metric experts get usage data from the EMI to monitor metrics and visualization usage. We already discuss most of the optimization needs from the metric experts in our discussion on the monitoring system (see section 6.1).

...wants to know and evaluate some technical details of the EMI. From our experience, most of the metric experts also have a technical background. Therefore, they want to be involved in some of the technical design of the EMI services. For example we evaluated and discussed database schema design and storage procedures of the metric kernels as well as data adapter design with metric experts because these implement the metrics.

...does not want to dig to deep into technical details of EMI services. Even though metric experts want to know some technical insides they do not want to be bothered with all technical details. Furthermore, they are typically not interested and do not have the knowledge and training to designing the actual EMI services.

...does not want to be responsible for the operation of the EMI. The metric experts want to monitor the functional usage and are responsible for the functional part (the metric portfolio) of the EMI. The technical operation, however, needs to be taken care of by trained operation personal. Thus, another important aspect that needs to be considered is to get actual operation personal "on board" to operate the EMI.

Responsibilities

The central responsibilities of metric experts in our process model are to ...

... manage and optimize the metric portfolio (by monitoring metric usage).

... design (monitor) prototypes to check metric system requirements.

... overview the implementation and maintenance of the EMI.

... define, control, and monitor metric related processes.

... aggregate and prioritize metric requirements.

... specify test cases for the EMI services.

... decide on the removal of metrics, services, and EMIs.

9.3.3. Architect

The responsibilities and needs of architects are covered in most software engineering process models and software development processes of the companies who use this engineering approach. Therefore we will only focus our description on some details regarding our process model and reference architecture. Most importantly in this regard is that the architects actually knows the reference architecture and are able to apply its concepts to instantiate the designs for the respective EMI services. Furthermore, the architects need to be familiar with the concrete architecture of the actual EMI under development in oder to make and justify design decisions.

The architect...

...wants to use reference architectures. The benefits of a reference architecture, such as our reference architecture for EMIs, is that they provide solutions for common problems and provide a common terminology among the development team. Using a reference architecture, therefore, speeds up the design process and eases the job of the architect.

...wants to have clear guidelines for architectural decisions. Most importantly, the architects need to make architecture decisions when instantiating the reference architecture. For example what data adapter pattern to use or what specific metric kernel design to choose. Furthermore, they also need to decide on the actual technologies to build the services. They, hence, need to provide a high number of decisions when instantiating the reference architecture. Therefore, they require guidance for these decisions, which our reference architecture already provides at key points with dedicated discussions.

. . . does not want to be restricted. Another important aspect is that architects are and need to be responsible for the design of the actual EMI. Therefore, they do not want to be restricted either by processes, governance, or the reference architecture. All these things are just tools to help the architect. In the end, however, the architects must be free to choose the best fitting solution for a particular problem in the particular environment!

Responsibilities

Architects are responsible for the actual architecture (metric services and their interaction) for an actual EMI. Therefore, their central responsibilities in our process model are to . . .

. . . design specific services based on the requirements from the metric experts.

. . . evaluate the design with metric experts, developers, and operators.

. . . provide assistance when maintaining services.

9.3.4. Developer

The developer role, similar to the architect-role, is covered in existing processes. With regards to our process model, developers need to be specialized in the area of developing EMIs. They, hence, need to know the reference architecture, its different parts, and the actual architecture of the EMI under development. Furthermore, they need to know specific frameworks, tools, and technologies required for the implementation of the actual EMI.

A developer. . .

. . . wants (and requires) clear and precise metric requirements and designs. The design from the architects and the requirements need to be clearly documented and fully provided to the developers. They guide and focus the construction of the actual EMI services. Furthermore, the developers need to be able to discuss certain aspects with metric experts, architects, and maybe even metric customers. Developers should also be included in the design activities (or at least the evaluation of the design).

. . . needs to be able to easily implement changes. In particular, the services should be design in a way that changes have minimal side effects. This is mostly covered by the microservice oriented design of the reference architecture using decoupled services.

. . . wants to use up-to-date technologies. The developers need to be able to implement the given design with whatever technology fits best and provides the least maintenance effort. Therefore, they need to be able to use up-to-date technologies

because most of them focus on solving particular issues. Furthermore, this also ensures that the developers like to work on the EMI because it gives them freedom. Nevertheless, this also increases the heterogeneity of the technologies and tools used in the actual EMI. However, a design goal of the reference architecture is to be able to handle this.

. . . does not want to develop additional glue-code. The development of an actual EMI should be based on a dedicated development platform for this EMI. This development platform should provide frameworks, templates, and technologies that help the developers to focus on implementing the actual business functions. Spending time with implementing glue-code or configuring services can be time consuming, boring, and the source of nasty errors.

Responsibilities

The central responsibilities of developers in our process model are to . . .

. . . implement and maintain (EMI) services.

. . . implements and maintain automated test cases.

. . . select suitable (sub) technologies for a given service or task.

. . . stage (deploy, test, and release) services in the different staging areas together with operators.

9.3.5. Operator

Similar to the developer and the architect before, the operator role is typically already defined in the existing processes in the company. Sadly, operation is vary rarely (specifically) covered in other process models and the needs of operators are rarely addressed in reference architectures and frameworks. However, from our experience, operation specialist who know the enterprise measurement infrastructure and participate in its design and construction are a necessity for the long time success of the metric system! Therefore, we included important services for the operators (like monitoring and logging) directly in our reference architecture. Furthermore, we also like to address the operation phase in our process model explicitly and provide best practices and solutions to common operation problems.

An operator. . .

. . . wants to see the status and detailed information of services. The job of the operator is to control that the services in and surrounding an EMI are working within their specified boundaries. Therefore, operators need dedicated tools to monitor the services. We already discussed a lot of important information needs from operators in the design of the monitoring system in our reference architecture (see section 6.1).

. . . wants a robust measurement infrastructure. The measurement infrastructure should be build in a way that a failure in one of its services does not result in a complete system failure. Our microservice based and loosely coupled reference architecture already provides a good framework to ensure the robustness of the actual EMI.

. . . wants to be included in the design and construction phase. The many success stories of dev-ops already provide a good justification to include operators in the design and construction phase alongside architects, metric experts, and developers [Hüt12]. Furthermore, this is also backed up by the experience in our field studies. All of the operators were very happy to be included in the development processes and provided valuable input to it. This also ensures the long time success of the actual EMI.

. . . does not want too much heterogeneity in the EMI. This contrasts the needs from developers and architects for including new technologies and building very heterogenic infrastructures. However, these infrastructures still need to be operatable and operators need to be able to include new services. Hence, they also need to be familiar with the new technologies. Nevertheless, new technologies also have the reputation of being unstable which jeopardizes the stability of the EMI. Therefore, stability needs to be taken into consideration when selecting the technologies and, again, operators need to be included in the design and development activities.

Responsibilities

The central responsibilities of operators in our process model are to . . .

- . . . ensure the availability of the EMI and its services.
- . . . restore a consistent state of the EMI after a failure.
- . . . ensure the EMI runs within specified performance boundaries.
- . . . initiate service redesign.
- . . . provide requirements for service redesign (split-up, merge, technology update).
- . . . provide and configure infrastructure services for an EMI (messaging services, databases, server nodes, application servers).
- . . . assist staging and release activities.

9.3.6. Role Involvement

The previous sections provided information about the needs and responsibilities for each role that is involved in our process model. In the section before we already provided rough overviews over the phases and central activities in each phase. This subsection will now provide the missing mapping between the roles and the phases and their activities. We experimented with integrating the roles into the process overview in figure 9.3 by the means of swimlanes. However, the figure significantly increases in size and was barely readable. Unfortunately, a figure this size does not fit easily into this thesis. Therefore, we choose to presents a workload overview of the roles in each of the phases similar to the overview provided for the unified process.

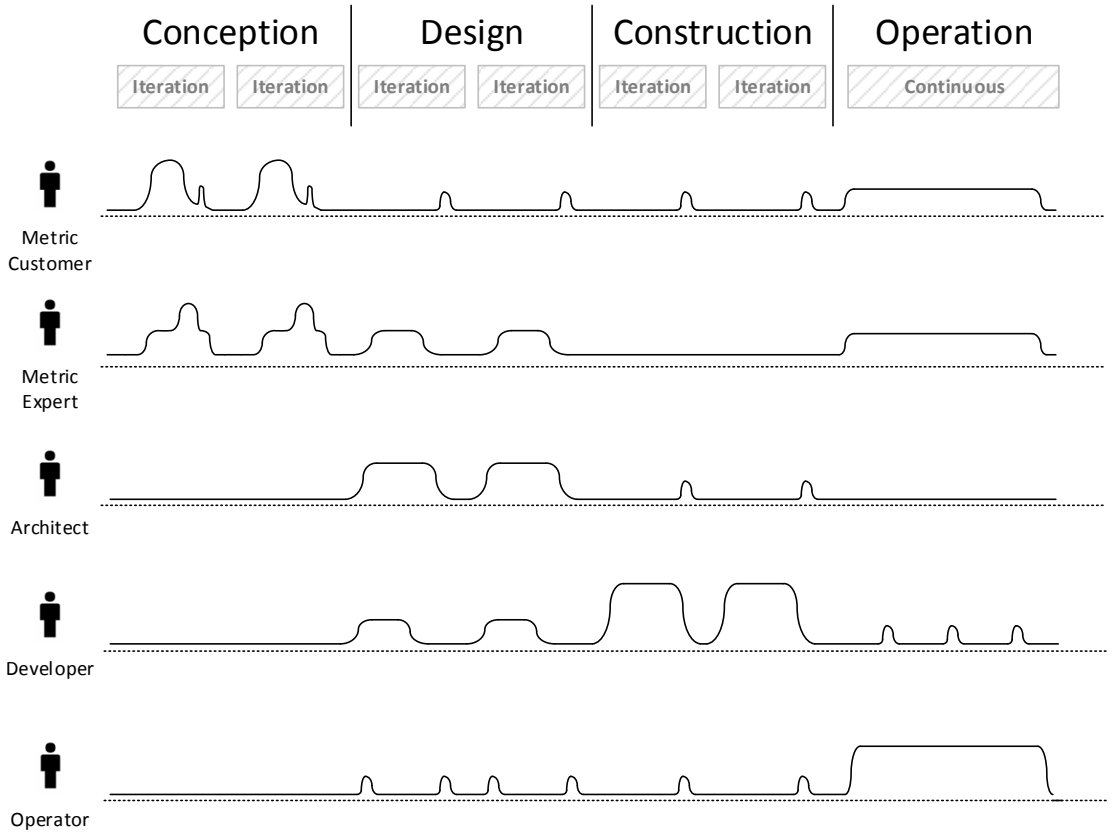


Figure 9.3.: Workload of the roles in the different phases of the development process

Figure 9.3 shows the workload of the roles in the different phases of our process model. The higher above the dashed line the indicators are the higher the workload of the specific role is in the particular activity at this point. Additionally, the top part also shows two iterations per phase to emphasize the iterative nature of the phases. The incremental nature of the process (especially in the design and construction phase) is not shown in this diagram.

In the conception phase naturally the load for metric experts and metric customer is

high because they need to gather the requirements for the metric system. The metric experts also need to prepare the prototypes, which are then evaluated by the metric customers. The metric experts also need to specify the iterations for the following design and construction phases.

The architects are constantly and heavily involved in the design phase. They are supported in the design by the metric experts and the developers. Both providing additional feedback and evaluation the design on the go. Additional, mostly non functional, requirements are added by the operators. At the end of each iteration the design needs to be evaluated by the metric customers and the operators.

The construction phase is of course dominated by the metric developers who are implementing the new and changed metric services. Once a service is constructed the implementation is evaluated by the architects against the design specification. After a successful evaluation the developers and operators stage the service, which can then be evaluated by the metric customer. If this evaluation is successful the service is released to the production system.

During the operation phase the metric customers use the metric system. The operators make sure the system is available and performs within the given parameters. The metric experts assist the metric customers in the usage of the metric system by providing training and additional documentation. They also constantly analyze the usage of the metric system gain a better understanding of the needs of the metric customers and find improvement potential which may trigger the next development iteration. The developers also need to occasionally maintain a service or fix a (small) bug in this phase.

This concludes the description of the different roles in our process model. The following section will provide additional information about the initialization of this process model.

9.4. Process Initialization

The last sections provided an overview over this process model and its central roles. This section will briefly introduce the initialization of the process model. The initialization contains additional activities that need to be carried out before starting the first conception phase. Most of these activities are taken from our experience from the field studies as well as measurement and metric best practices.

Get Manager Commitment Mangers provide the resources for the metric initiative including the construction, maintenance, and extension of the measurement infrastructure. Manager commitment is also listed as an important success factor for metric initiatives [NV01, HMO08]. Thus, it is important to get their support as early as possible. A good way, therefore, is to advertise the advantages of a well design metric system and clearly defined roles and responsibilities in a specific metric system engineering process as well as a flexible and easily evolveable measurement infrastructure.

Provide Training Before starting the construction of an EMI and initiating the first conception phase it is important to train the metric experts, the architects, the developers, and the operators. The training should include training sessions on instantiating and using the reference architecture as well as detailed discussions on the process model.

Infrastructure Setup Metric Operators and Metric Developers need to setup the required infrastructure to enable development. The following checklist provides some important aspects of the infrastructure setup:

- ☐ Setup web-, application-, and messaging-server for the different staging environments.
- ☐ Setup development environments on the development machines.
- ☐ Setup source code repositories for the artifacts.
- ☐ Setup artifact management systems for the releases.
- ☐ Setup continuous integration infrastructure for automatic build and test of the services.

Instantiate the Process Model This process model, as the name suggests, is just a model. Therefore, it needs to be instantiated to an actual process before using it. Most importantly, the instantiation needs to ensure that it reflects the existing development processes in the actual company. Otherwise, people will not be familiar with the activities and it will be strange and artificial. Furthermore, the templates for the artifacts required in the process need to be defined in a way to match the company policies (CI, tracing information, and so on). The appendix of this thesis provides help for the actual definition of the templates.

Also the technical processes need to be defined. Most importantly, the development, the staging, and the deployment processes (for example the staging levels). This also need to refer to the tools and infrastructure provided at the company.

Initial Construction and Development Before the implementation of actual metric services all mandatory services for the EMI need to be constructed. Most importantly, this includes the services in the two integration layers (see section 5.2 and section 5.3). Furthermore, it also includes the operation services like the monitoring service, the logging service, and the directory service (see section 6).

Another important aspect is to construct the development platform for the EMI. This needs to include frameworks and libraries to easily connect to the different parts of the EMI and easily integrate the services with the operation services. Section 5 about the technical reference architecture provides a lot of hints on components and parts that should be included in the development platform.

After performing all these activities the actual first phase of the process can start. We provide additional details on the activities in this conception phase in the following chapter.

10
The Conception Phase

The conception phase is a classical requirements engineering phase. The purpose of metric systems is to satisfy information needs of metric customers. Hence, the most important activities are to gather these information needs and provide meaningful answers.

Metric projects typically span across the whole organization and include a large variety of stakeholders. Thus, once the metrics are implemented and rolled out into production they can not (and should not) be changed easily. Therefore, typical fast agile approaches with short increments that lack an explicit prototyping phase and only include a short requirements phase are not applicable.

We provide a lot of details on the activities in this phase. The reason for this is two folded. First, we realized that it is important to get the information needs, monitor prototypes, and increment plan right before starting the development increments. Therefore, we provide this high level of details to ensure that all of the important activities are performed. Second, a lot of the activities in this phase are independent of the instantiation of the process model. Contrasting, for example, the construction phase which contain a lot of hard to grasp parallel activities. Therefore, this chapter is the largest of all of the chapters describing the process phases.

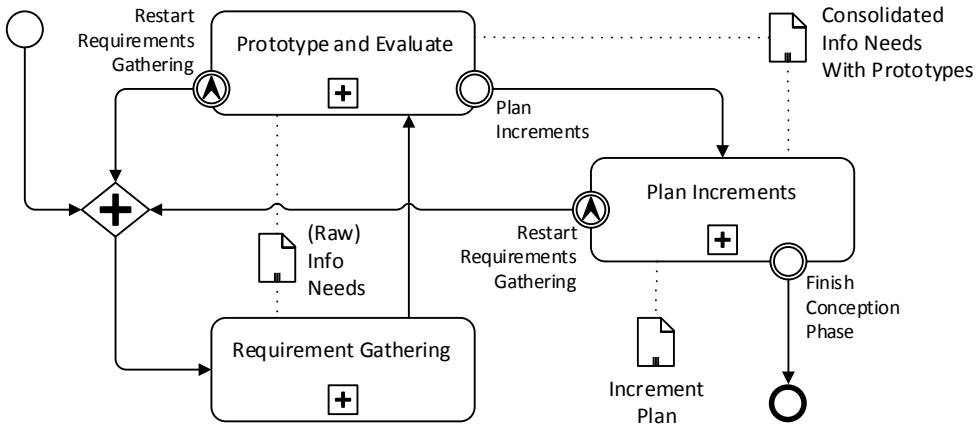


Figure 10.1.: Conception phase overview

Figure 10.1 provides a BPMN diagram for the rough overview over the three core activities in the conception phase: **`Requirements Gathering`**, **`Prototype and Evaluate`**, and **`Plan Increments`**. The activities are inspired by the GQM method. However, we extend it with some, in our view, key aspects like prototyping and evaluation.

The general idea behind the activities is to first get the information needs from the metric customers as questions based on their goals. Then metric experts will develop monitors and metrics that will answer these needs. The monitors are then included in prototypes which are evaluted with the metric customers. If they are not satisfied, these steps are reiterated. Otherwise, the monitors and metrics are integrated into the `Increment Plan`, which organizes the incremental design and construction after this phase.

The requirements analysis starts with the organization and analysis of the information needs that were gathered before (either as new information needs or as changed information need). We strongly recommend organizing them in a quality model starting at the top level information needs (or goals) and breaking them down into lower level information needs. Furthermore, each monitor also requires a definition for the visualization. According to Few [Few12, Few06] the metric expert should keep the visualizations as clear and simple as possible to increase their readability. However, some (visualization) designs are sometimes hard to grasp. Hence, these need to be evaluated with the metric customers. The evaluation can of course include training the metric customers to read certain new, unfamiliar, or unpopular visualizations.

One of the main goals of this phase is to develop a (horizontal) prototype which shows the complete visualization and can be evaluated with the metric customers. We already discussed the reasons to integrate prototyping in the development process. However, we would go one step further and argue that not including prototyping during the development of a metric system is careless and will inevitably lead to problems. Prototyping will help to identify these problems as early as possible; hopefully before spending a lot of money and time with the construction of the metric system.

Paper prototypes or special dashboard prototyping tools should be used to construct the prototypes because the prototypes will be changed often and the metric experts need to be able to quickly apply changes. Additionally, these tools help to provide prototypes that look like prototypes with sketchy lines and an unfinished look. This is important in order to avoid misunderstandings of the construction state of the metric system.

If the data provision mechanisms for certain metrics are complex then the calculation algorithms and signal chains should be prototyped in a vertical prototype as well. These prototypes can later be used as the basis for the incremental construction of the specific services in the EMI.

The following sections provide additional details on the three main activities. The following section 10.1 will discuss the two requirements gathering techniques: workshops and interviews in greater details including their strengths and weaknesses. Section 10.2 will then give additional details to the construction of the prototypes and the general evaluation of the information needs with metric customers. In the end, section 10.3 provides additional details on the **`Plan Increment`** activity. We finish the description of the construction phase with a short summary in section 10.4.

10.1. Requirements Gathering

The goal of this activity is to get a (weighted) tree of information needs for a group of (or individual) metric customer(s). The weight of the information need should reflect the importance of the information need to the metric customer(s). It is later used for planing the iterations and increments.

Including metric customers in the process and making them the source for the metrics is one of the most crucial aspects of our process model. Umarji and Seaman developed a metric program at a large international organization [US08, USE]. They list the inclusion of the developers (their metric customers) as a crucial success factor. Furthermore, Fenton and Hall also identified this as an important factor [HF97]. It is also important that the analysis and interpretation of the metrics is publicly availible and based on documented interpretation aids as identified by Dekkers [Dek99], Seibert [Sei03], and Pfleeger [Pfl93]. Also see our discussion on related literature in the introduction in section 1.2.1.

Different methods can be used in order to gather the information needs from the stakeholders. The two main methods we recommend are interviews and workshops. Both have their strengths and weaknesses. On the one hand, interviews provide a more personal set of information needs for a particular stakeholder but require an intensive treatment of the combined information needs lists from different interviews. On the other hand, workshops by their nature provide a more unified list of information need but some important (individual) information need may get lost. Regardless of the method other useful requirements gathering and structuring techniques should be applied in the interviews and workshops. The actual process, that is instantiated from our process model, needs to state what combination and in what order the different techniques should be combined (see advantages and weaknesses of the two in the following subsections).

10.1.1. Activity Overview

Figure 10.2 provides an overview over all the subactivities in the Requirements Gathering activity as BPMN diagram. The activity starts on the top left and from there enters the **`Plan RE (Need Identification)`** activity. This includes analysis activities which provide a first list of opaque information needs which are defined later on. It also (re)defines the `RE Plan` which guides the requirements engineering activities and steps in the next activity. This next activity is therefore, called **`Execute RE Plan`**. It contains a lot of subactivities itself. Most importantly we distinguished between two requirements gathering techniques: workshops and interviews. Both have their specific preparation activities and specific preparation artifacts (`Questionnaires` and `Workshop Slides`). The output of both techniques is a set of `(Raw) Information Needs`. These are sorted, enriched, and clearified in the following **`Process Result`** activity. From there the metric expert can start another iteration of the processes if the `RE Plan` defined multiple iterations or additional information needs from additional metric customers need to be found. The reiteration either starts directly or can optionally redefine the `RE Plan` in order to reflect the new information. The following subsection will provide additional details on the first activity: **`Plan RE`**.

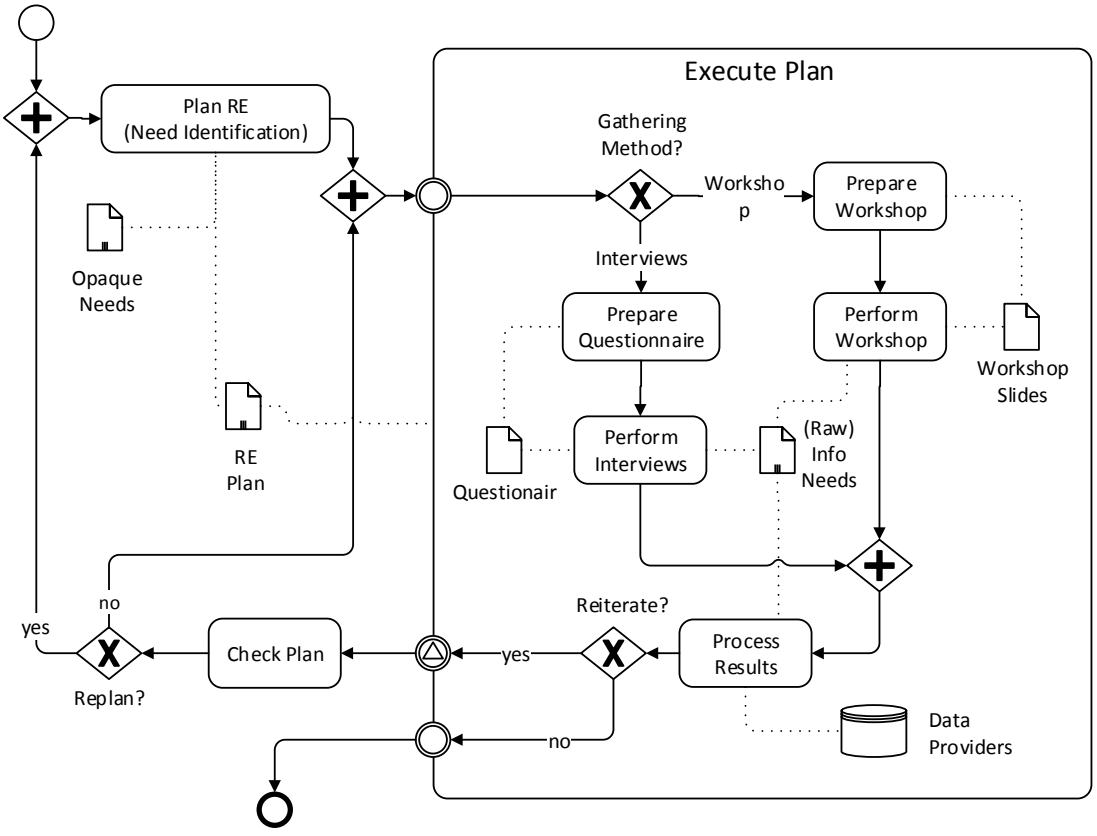


Figure 10.2.: Requirements Gathering activity as BPMN diagram

10.1.2. Plan Requirements Gathering and Information Need identification

The goal of this activity is to plan the requirements gathering activity based on the initial information from the metric customers. Therefore, the output of this activity is a plan for the requirements gathering (the `RE Plan`) as well as a list of first opaque information needs. Most importantly, this activity includes the analysis of the existing processes that the metric customers are using as well as the tools that are used by them. The result of the analysis will then drive the plan and the gathering techniques used.

The `RE Plan` also includes iterations and goals for each iteration. An example could be to first perform a workshop to identify goals and coarse grained needs from the metric customers. These needs can be refined in a second workshop. To get a better understanding of specific needs a third iteration can perform selective interviews with specific metric customers.

The first important step is to analyze the processes that the specific (set of) metric customer is involved before starting the interviews or workshops. This helps to understand specific terms the metric customers might be using during the interviews or workshop sessions, which avoids misunderstandings. Furthermore, the metric experts can analyze potential data sources (artifacts that are used in the process) before starting the interviews

or workshops. This analysis may point the metric experts to obvious information needs or optimizations in the process, which they can present in the interviews or workshop sessions.

Typically information needs originate at specific points in the process which are easily found if the process is being visualized. We recommend to visualize the process during this analysis phase if no visualization of the process exists or use the existing visualizations. An additional side-effect is that the result of this analysis (process diagrams) can be used in the interviews and workshops as additional sources for information needs if the processes are not (well) documented.

The other important analysis step is to analyze the tools that are used by the metric customer because these tool are very likely to become data providers in the measurement infrastructure. Furthermore, the the data models insight the tools provides additional insights into the processes. They may also indicate certain visualization opportunities which can later be included in the prototypes.

After the `RE Plan` is defined in this section the process enters the main big part of this activity: the **`Execute Plan`** activity, which we will further discuss in the following section.

10.1.3. Execute RE Plan

The overview diagram in figure 10.2 already indicated the importance of this subactivity. It is the core of the **`Requirement Gathering`** activity. The goal is to gather information needs from metric customers by executing the steps defined in the `RE Plan` in the last subactivity. We differentiate two requirements gathering techniques for gathering information needs from the metric customers: Interviews and Workshops. The following subsection provide additional discussions on the advantages and weaknesses of each of these techniques in our context.

Workshops

The general idea of a workshop is that a group of metric customers identifies their information needs together. Workshops can therefore be used to gather a lot of different information needs for a specific group of metric customers. The following two lists provide the key advantages and weaknesses of workshops for gathering information needs from our experience in the field.

Advantages

- ✓ Workshops provide on-the-spot discussion about the needs and their origin the goals and subgoals with different people within the same group of metric customers.
- ✓ The metric customers influence each other which provides additional input and different perspectives to the different needs.

✓ A broad discussion in the first workshop can open-up new fields which then guide follow-up iterations.

Weaknesses

✗ It is hard to cover individual needs of single metric customers which could be important.

✗ Workshops need to be moderated to avoid too much discussions off-topic,

✗ Metric customers influence each other which may cloud certain needs from particular customers.

✗ A lot of information needs and goals are discovered in all workshops and it can be hard to satisfy them all. This requires clear communication all the time about what is going to be addressed and what is postponed to later iterations or increments in order to avoid dissatisfaction later on.

✗ Preparation of the workshop with specific slides is time consuming for the first workshop session. However, the material can later be reused for other workshop sessions.

✗ Scheduling appointments with multiple metric customers may be difficult. This gets harder the higher up in the organization structure the metric customers are located (managers are harder to schedule than developers).

Interviews

In interviews single metric customers are interviewed to get their individual information needs. The result of multiple interviews with similar metric customers can then be integrated to form a rough idea about the information needs of a group of metric customers. However, these integrated information needs need to be discussed with the metric customers to check if the selection and the importance of the needs are correct. The following two lists provide the key advantages and weaknesses of interviews for gathering information needs from our experience in the field.

Advantages

✓ Interviews provide clear and in depth insight into the information needs of single metric customers.

✓ Metric customers tend to provide more insights if they are not in a group of people and may also provide “of the record” information which may come in handy later on.

✓ Scheduling the appointments with single metric customers is typically not as hard as trying to schedule multiple metric customers.

Weaknesses

- ✗ The information needs of the different interviews need to be integrated and discussed with the metric customers after the interviews. This is a lot of additional effort!
- ✗ The preparation with specific questionnaires is time consuming and requires additional discussions between the metric experts before performing the interviews but it increases the quality of the interview and the discussion significantly!
- ✗ Interviews provide no on-the-spot discussion between different points of views in a group of metric customers.

Most of the time we started the requirements gathering in our case studies with one to four workshops to get a broad set of information needs and let the metric customers influence each other (for good). We then utilized interviews in later stages or to get specific sets of information needs. We provide additional guidelines for the actual execution of the two techniques, based on our experience from our field studies, in the appendix in section B.1.1.

10.1.4. Process Results

After these information need gathering activities the results, huge trees or lists of information needs, of the interviews and/or workshops need to be integrated, sorted, and cleaned, which is performed in the **`Process Results`** activity. The information needs are integrated into the `Raw Information Need` list and potential data providers (tools) are registered in the `Data Provider` repository. However, it is not the focus of this activity to integrate these new information needs.

10.2. Prototype and Evaluate

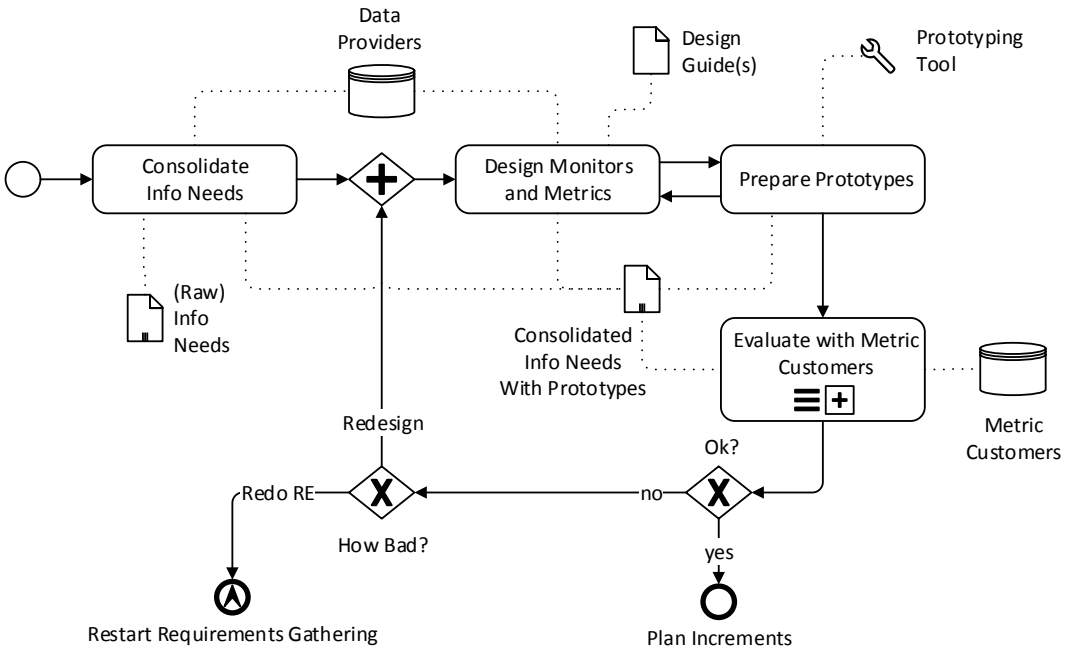


Figure 10.3.: Prototype and evaluate activity as BPMN diagram

As the name suggests the focus of this phase is to evaluate the requirements with the metric customers. Like in every other software development project this activity is very important to avoid misinterpretations between (metric) customer and development team. The activity itself contains four mayor subactivities which are performed sequentially. Figure 10.3 contains an overview over these subactivities and the artifacts as BPMN diagram. First, all the raw information needs gathered in the previous activity are consolidated. Second, the monitors and metrics are designed by the metric expert and most importantly, prototypes of the monitors and maybe of the calculations are constructed. Third and finally, these prototypes are evaluated with the metric customers. The following subsections provide additional details on these subactivities.

10.2.1. Consolidate Info Needs

The output of the previous activity is a raw list (or tree, or mind-map) of information needs from different metric customers. This activity needs to consolidate these raw needs from the different metric customers to one or several focused sets of information needs. Building these sets and consolidating the information needs is important to get a consolidated overview over all the information needs of the metric customers.

In order to consolidate the information needs, metric customers may need to:

- Combine several slightly different information needs to one information need.
- Extract several (more focused) information needs from a raw information need.
- Reorganize and maybe reformulate information needs to fit a group of metric customers rather then specific information needs for one particular metric customer. This may also include introducing additional information need categories or root questions.

Additionally, the metric experts should check, create and update trace links between information needs and potential data sources in the data provider repository during this activity.

10.2.2. Design Monitors, Design Metrics and Prepare Prototypes

A depiction of the planed monitors and/or dashboards is more easy to grasp for metric customers then plain text descriptions. Hence, they are more likely to provide important feedback before the construction. Therefore, it is important to create horizontal prototypes of the dashboards and monitors after an initial gathering of information needs. These prototypes are used to evaluate the proposed monitors (solutions) with the metric customers. In order to focus the metric customers on the evaluation of the monitors and dashboards it is important to keep them *sketchy*. Furthermore, this will avoid confusion between the real implementation and the prototypes. Thus, we recommend to use specialized tools to build these prototypes[1].

We also recommend to base the prototypes on typical data and not just random values to show the metric customers what an actual monitor for their entity of measurement may look like. Optionally, the metric experts can prepare different prototypes for different situations; for example: a good project, a typical project, and a bad project. Often, the theme of the information needs is to know if everything works as planed or if something is strange. The different prototypes for the different situations help the metric customer to evaluate whether they are able to identify these strange situations. Furthermore, the different prototypes can later be used as part of the interpretation aids for the monitors.

The design of the monitors and metrics should be based on design guides which are established in the company. As a good starter for the design of monitors we recommend "Show me the Numbers" by Stephan Few [Few12]; for the design of dashboards we recommend "Information Dashboard Design" also by Few [Few06]. As a general role of thumb: The monitors should be as simple as possible and information should not be encoded in arcs. Therefore, stacked-bar-charts and pi-charts should be avoided if possible. Also colors should only be used very sparsely to indicate certain important aspects (for example warnings or odd data).

[1] A discussion on the development of a specialized dashboard prototyping tool is provided in the bachelor thesis of Matthias Gora [Gor13].

10.2.3. Evaluate With Metric Customers

This is one of the key activities in the conception phase. As the name suggests the goal of this activity is to evaluate the prototypes with the metric customers from the **`Requirements Gathering`** activity. It is important to get as much feedback to the suggested monitors and dashboards as possible. Because in this phase it is still relatively cheap to change all aspects of the monitors. The metric customers can choose among a large variety of methods and tools to gather the feedback. We used: workshops, interviews, mail + response, or web collaboration systems like forums or wikis in our field studies. The following list contains a quick discussion on the benefits (✓) and weaknesses (✗) of each of these approaches.

Workshops and Interviews

- ✓ Explanations for the monitors and dashboards can be given face-to-face, all other forms require additional documentation and explanation, and documentation may be ignored leading to unproductive feedback.
- ✗ The face-to-face explanation can influence the metric customers leading to little feedback. Therefore, problems with the proposed solutions may be overlooked.

Mails

- ✓ Quick, easy to distribute, and metric customers are used to this form of communication.
- ✗ No (direct) way of personal communication, metric customers may ignore the mail, and additional effort for integrating the feedback from all metric customers.

Web Collaboration Systems

- ✓ Some form of (indirect) personal communication possible, relatively quick and easy, and feedback is located in one place easing the integration.
- ✗ Metric customers may not be familiar with these system and answers typically visible to everyone which may lead to less feedback.

The actual activities and steps that are performed in this subactivity need to be defined when instantiating the process model based on the processes in the company and company policies. This leafs less room for interpretation and eases the execution of the process for all the stakeholders.

From our experience, the optimal evaluation of the proposed monitors and dashboards with the metric customers combines a workshop or interviews with mails and a web collaboration system. The metric expert starts by sending a mail with the prototypes of dashboards and monitors plus additional documentation to the metric customers.

In parallel the metric expert prepares a discussion platform in the web collaboration system containing the same information. The link to this platform should also be included in the mail. Additionally the metric expert should prepare a workshop with the metric customers or prepare interviews with selected metric customers. Because there is typically some time between the mail and the workshop or interviews the metric expert can start integrating some feedback from the metric customers from the web collaboration system into the prototypes of the monitors and dashboards. Therefore, the workshops and interviews are a second iteration which helps to faster reach a feasible solution. Because the workshop or interviews are direct interactions between metric customers and metric experts this also helps to get informations from metric customers who did not participate in the discussion in the web collaboration system.

After the evaluation the metric expert needs to gather all the changes and decides how to proceed. Small changes can be performed directly. Minor problems are solved by redesigning some monitors, metrics or parts of the dashboard. After the redesign the prototypes need to be evaluated again to check the changes with all metric customers. If the evaluation shows major misunderstandings between metric expert and metric customers, however, the **`Requirements Gathering`** activity should be restarted.

10.3. Plan Increment

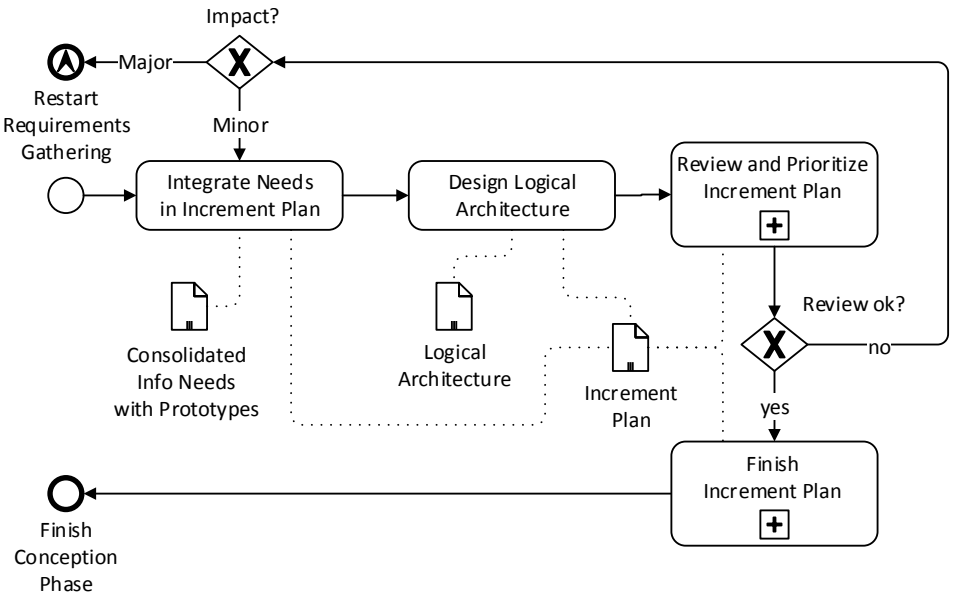


Figure 10.4.: Plan Increment activity as BPMN diagram

One of the core concepts of the metric management process model is its incremental development core. However, these increments should not be build add-hoc but systematically planed. Thus, the goal of the **`Plan Increment`** activity is to provide an `Increment Plan` for the following phases. This plan can, of course, be changed after every iteration; for example when the environment changes or the priority on the information needs are reassessed and changed. Figure 10.4 provides an overview over the subactivities in this activity. Similar to the **`Prototype and Evaluate`** activity, the **`Plan Increment`** activity contains four subactivities which are executed sequentially. First, the **`Integrate Needs in Increment Plan`** subactivity, as the name suggests integrates the information needs gathered and evaluated before into the `Increment Plan`. Second, from this the architect and the metric experts develop the `Logical Architecture` for the EMI, which contains the metric applications that typically make up one increment (see section 4). Third, the `Increment Plan` is reviewed and prioritized. Forth, if the review is successful then the **`Finish Increment Plan`** subactivity finalizes the plan, finishes the conception phase and starts the first development increment and the first design phase. The following subsections provide additional details to each of the core subactivities.

10.3.1. Integrate Information Needs and Design Logical Architecture

In this subactivity the consolidated and evaluated information needs together with their respective prototypes need to be integrated into the increment plan. As the name suggests

the `Increment Plan` lays out the road map for the incremental design and construction of the EMI based on the given information needs. An increment, in general, should provide a benefit for some metric customers (i.e. answer some of their information needs)! Furthermore, the increment should be coherent (see section B.1.2 in the appendix for a guide on coherent increment plans). The integration of the new information needs may also change existing increments that are already planned.

After the increments are planned the metric experts and architects should define the `Logical Architecture` of the EMI. The logical architecture defines the logical components that are required in an EMI to satisfy the given information needs. The logical components are organized in metric applications (see section 4.1). Metric applications are coherent sets of logical components that solve a specific purpose. Therefore, this is closely related to the increment plan and often the metric applications reflect the development increments.

10.3.2. Review and Prioritize Increment Plan

After the increments are planed and the logical architecture is defined the metric experts need to (re)prioritize the increments. The order (their priorities) of the increments may depend on several aspects which are often company specific. Hence are not listed here.

After the prioritization, the increment plan needs to be reviewed by the architects and the operators to avoid complications during design, construction, and operation. Selected metric customers should also be included in the review to integrate them in the planning process once more. The plan can also be send out to all metric customers or made available via an web collaboration system to allow for more feedback. One possible outcome of the reviews of the increment plan might be that the plan is rejected. Maybe these problems can be solved by small changes and reiterating the evaluation. And maybe the whole requirements gathering process needs to be restarted because some metric customers or needs were completely overlooked.

10.3.3. Finish Increment Planing

When the review of the `Increment Plan` turned out positive (or with minor changes) and all stakeholders accepted the plan it can become final. Consequently, it should be send to all stakeholders or made publicly available inside the organization. This also marks the end of the conception phase and the process can enter the incremental design and construction phases which follow the `Increment Plan`.

We just provided very brief descriptions and discussions on these subactivities. However, figure 10.4 already suggests that the two activities **`Review and Prioritize Increment Plan`** and **`Finish Increment Plan`** need to be further defined when instantiating the process model. For example all the actual review activities and their flow as well as all the detailed activities when finishing the conception phase need to be defined. These activities can become large but we recommend to model them explicitly when instantiating the process model, because this makes the process more specific which eases the execution of the process for all the stakeholders.

10.4. Conception Summary

This chapter presented details on the conception phase of our metric systems engineering process model. We first provided an overview over the activities in this phase and discussed related work. We then provided additional details on the activities and the core artifacts. The **`Requirements Gathering`** activity contains the core requirements elicitation activities. We discussed the differences and benefits of the two main techniques: workshops and interviews. The results from the gathering activity is a raw list of information needs which are consolidated in the next activity: **`Prototype and Evaluate`**. As the name suggests, this activity also contains the prototyping subactivities. Constructing and using prototyping in this metric related context separates our process model from existing approaches. Additionally, from our experience, providing and discussing prototypes with metric customers provides a lot of important feedback early on in the process. At this point metric experts are still able to easily (and cheaply) change some of the monitors and metric if they see that the expectations from metric customers are not meet. Finally, the **`Plan Increment`** activity produces and evaluates the `Increment Plan`, which guides the following design and construction phases. The following chapter will provide additional details on the activities in the design phase.

11
The Design Phase

As the name suggests, the goal of the design phase is to design and specify the services and the tests for the EMI required for the specific increment (for the specific metric application). During the design phase the `Design` artifact is created and evaluated. This provides the guideline for the following construction phase and contains all the results from this design phase. Most importantly, the design contains the actual architecture of the EMI which instantiates our reference architecture.

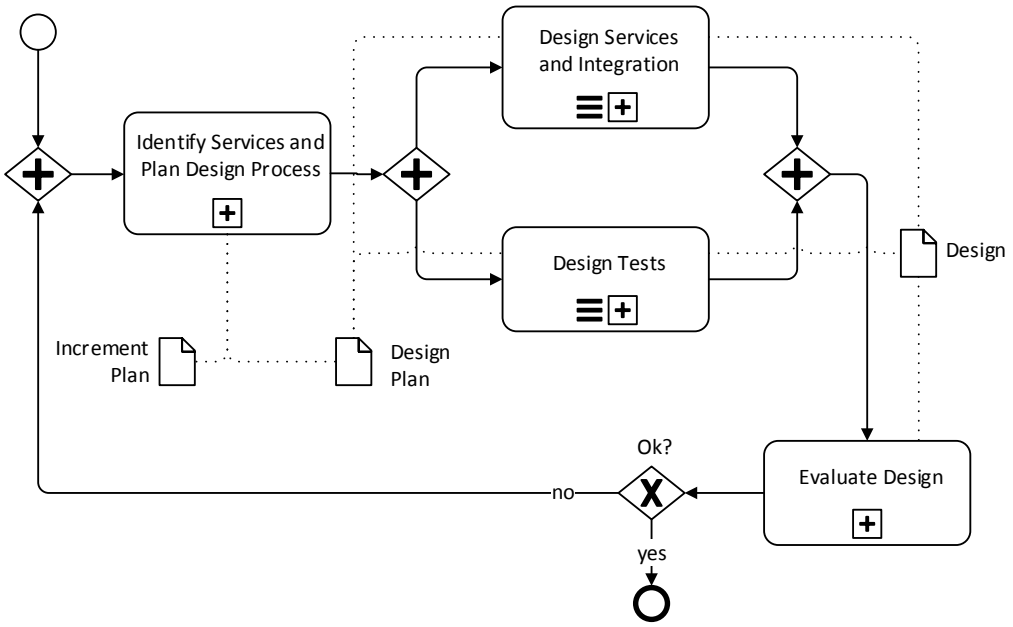


Figure 11.1.: Design phase overview as BPMN diagram

Figure 11.1 provides an overview over the core activities in the design phase. The phase starts with the **`Identify Services and Plan Design Process`** activity. The activity takes in the plan for the current increment. Based on the information needs and prototypes in the `Increment Plan` the metric expert and architect define the `Design Plan`. This contains the detailed activities that should be performed in the following activities. We differentiate two core design activities: **`Design Services and Integration`** and **`Design Tests`**. This emphasizes the importance of the test design because it is separated into its own activity. Both activities are executed iteratively and incrementally in parallel to each other mainly by the architects following the `Design Plan`. They both produce the `Design` for this increment which can contain new services, service migrations, service replacements, and service removals. The `Design` is evaluated by metric experts, architects, operators, and maybe metric customers in the last activity of this phase. The following sections provide additional details on these activities and the design itself.

11.1. Identify Metric Services

The first most important activity in the design phase is to identify the metric services which need to be changed or created in this increment. The requirements (information needs, metric customers, and monitor prototypes as well as metric definitions) for the metric services are specified in the `Increment Plan` for the current increment. This activity is performed by metric architects assisted by metric experts

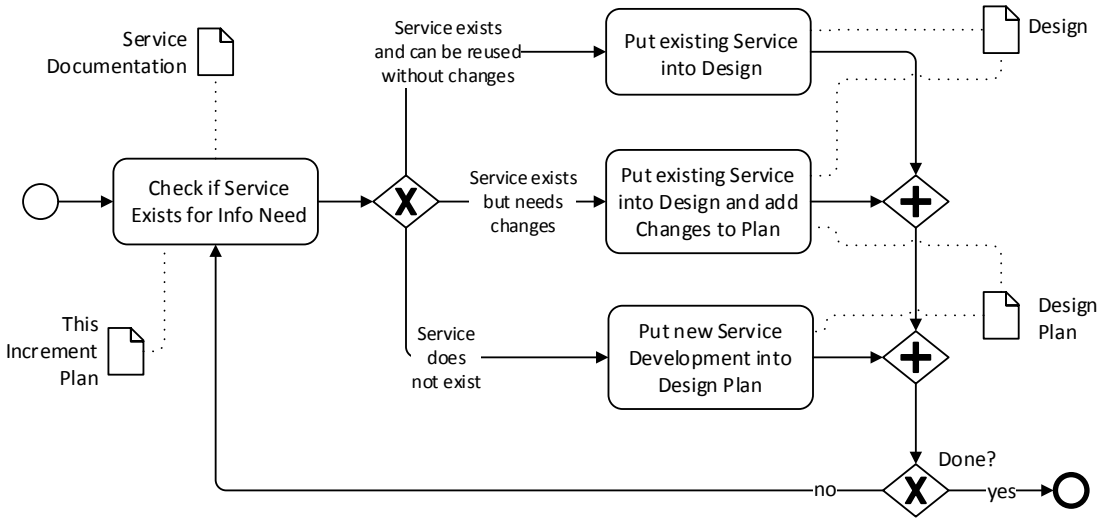


Figure 11.2.: Metric Service identification activity as BPMN diagram

Figure 11.2 provides an overview over the subactivities and artifacts in this activity. The first step of the identification is to identify which metric services (data adapters, metric kernels, visualizations, visualization frontend, and operation services) are required

for this increment and whether they already exist or not. Similar to other development processes and following the spirit of our reference architecture, the general intention should be to reuse as many existing services as possible. This may require some refactoring in the actual service to add the new functionality. Based on the reuse decision the following activities design the service and fill the `Design Plan` and `Design`. If an existing service can be reused without changes then the service can be put into the `Design` without adding additional steps to the `Design Plan`. If the service can be reused but requires changes then the service can be put into the `Design` and redesign activities need to be added to the `Design Plan`. If no existing service can be reused then only this new design activity needs to be added to the `Design Plan`.

We already discussed reuse of (EMI) services in our reference architecture (see section 4.1) and the foundations (see section 2.2). However, we would also like to provide some remarks here to assist the actual decision process for deciding to reuse an existing service or not. On the one hand, avoiding new metric services helps to deal with increasing structural complexity. On the other hand, it increases documentation effort for the reused metric services. This additional effort could also be used to created additional documents and overviews to deal with increased structural complexity. Furthermore, reusing existing services may reduce robustness of the whole EMI because few services become more important rather then distributing the responsibilities across different services in the infrastructure. Additionally, reusing the existing services is not necessarily cheap because typically some things need to be tailored to fit the new need of the service. As we already discussed, reuse is only feasible if less than 20% of the existing service needs to be changed. However, existing metric services are likely to have a better overall quality than new metric services because they are already tested and running in the field. The appendix contains a very thorough checklist in section B.2.1 to aid the reuse decision process, which we compiled from our experiences in the field.

If required this activity can also contain additional requirements gathering activities if specific parts in the increment plan require clarification. We did not model this explicitly in the overview figure because it would require to many decision nodes and transitions. However, the actual process should contain at least explicit jump marks and aboard criteria for this activity which may result in an aboard of the design phase and lead back to the conception phase depending on the severity of the problem.

11.1.1. Setup the Design Plan and Design Document

During this activity the `Design Plan` and `Design` document need to be setup. This subsection contains some additional remarks on the setup of the two artifacts.

Like the name suggests the design plan is a plan for the design phase. It hence should contain the metric services that need to be (re)designed in order to satisfy the requirements in the increment plan. It should also contain the test plan for the new or changed services (test objective, test identification procedures, and test end criteria). The design of the test is also included in the design document for this increment. Last but not least the design plan should include a plan for the evaluation of the design (who evaluates, what, and when under which objective).

In this activity the architect and metric expert should add emerging changes and pre-designs to the `Design` document. Typically the first rough overview diagram for the actual architecture is created in this activity as well. We provide a lot of additional details on the actual sections of the `Design` document and their setup and content in the appendix in section B.3. The following two sections provide additional details on the actual design activities which fill the `Design` document.

11.2. Design and Evaluate

The following subsections provide additional details to the core subactivities of the design phase. We kept the information as simple and as sparse as possible in here and moved a lot of the actual guides, best practices, and artifact descriptions to the appendix.

11.2.1. Design Services and Integration

Designing the services and the integration between them goes hand-in-hand and is the core activity in the design phase. This section contains a very brief discussion on the actual subactivities. The details are found in the description of the `Design` document in the appendix in section B.3. Furthermore, we also provide finer detailed design guidelines for the different types of services in the appendix in section B.2.2.

Most importantly, the design of the services and the integration should obviously instantiate our reference architecture from part II; more specifically our technical reference architecture in section 5. Therefore, the design of the services should follow the microservice architectural style. Therefore, the technologies used in each service can be specifically tailored towards the needs of the particular service. Most importantly, this includes the database and persistence technology for the metric kernels (see section 5.5).

The integration between data adapter and metric kernels should follow the concepts of the enterprise measurement data bus (EMDB – see section 5.2). The integration between metric kernels and visualization frontends should follow the enterprise uniform metric kernel access design (EUrEKA – see section 5.3). In general, the integration between the different parts should follow the idea of contracts between the different parts. These contracts should be the driving force for the design of the service. The services and its integration should be design in a way that they fulfill the contracts in all situations[1].

The architecture overview diagrams used in the `Design` document (and our reference architecture) use a combination of UML component diagrams and EIA pattern notation, see Enterprise Integration Patterns by Hohpe and Wolf for further details [HW03a]. The EIA pattern notation is added to the UML component diagrams in order to design topics, messages, and enterprise service bus items.

[1] For example a valid and consistent message send over the bus needs to be accepted by the corresponding metric kernels (always!). Even if from the view point of the metric kernel the data is not consistent (for example if the data is too old). In such a situation the metric kernel can always avoid processing the data in the message and enter the out-of-sync state. However, it must accept the message!

Another important step in the service design is to define the performance indicators and their boundaries for each service. These indicators need to be integrated and controlled via the monitoring system of the EMI. Hence, it is important to include operators in the design as well. They may also add other important information that may improve the design of the actual service.

11.2.2. Design Metric Service Tests

The design metric services and integration and the design metric service tests activity are often performed in parallel. The metric architects may (and will) fluently switch between the two activities. Especially the design (and test) for exception behavior calls for this switch because every exception needs to be thoroughly tested.

Regression tests by the means of unit and integration tests ensure less stress during construction of the EMI. Furthermore, system tests provide a formal specification of the core functionality of the services. They also act as large scope regression tests that ensure the overall performance of a defined scenario in a metric system.

Further details on testing[2] are provided in the design guidelines in the appendix in section B.2.3

11.2.3. Evaluate design

The `Design` document created in the previous activities is the basis for the construction of the system. Therefore, we need to make sure that it meets expectations and contains all important aspects. The previous sections already mentioned some of our statical quality assurance tools (guidelines, checklists, and best practices). However, we also recommend analytical quality assurance measurements to ensure that the `Design` document is checked one last time before entering the construction phase.

We recommend to perform a formal review (inspection) on the `Design` document on this subactivity. The guides and checklists mentioned above can act as guides for the review. Furthermore, we recommend to include developers, metric experts, and operators in this review as well because they have different expectations from the design. Also, metric customers may join the review in order to check whether the services proposed are able to answer their information needs.

This already concludes our very brief discussion on the design phase of our engineering process model. The following section will provide additional details on the construction and the operation phase.

[2] Additional details on actual test pattern and test templates for EMI services are provided in the bachelor thesis of Marco Moscher [Mos14].

12
The Construction and the Operation Phase

This section provides additional details to the construction phase and the operation phase of our process model. We combined the two phases into one chapter because we do only provide very little details to the construction phase. The reason for this is the strong influence of the actual technologies and company processes on the activities in the construction phase and its dependence on the design and increment plan defined before. Most importantly, the construction phase contains all the staging activities. Hence, we will discuss the minimal stages to support the different test levels for services in an EMI. The following section 12.1 provides additional details to the activities in this phase. After that we provide a lot of additional details on the activities of the operation phase in section 12.2.

12.1. The Construction Phase

The construction phase realizes an increment of a metric system according to the `Design` document created in the design phase. The construction activities will, thus, alter an EMI which provides the data for the visualization frontends and includes the calculation of the metrics in metric kernels as well as the data adaption.

The actual implementation activities may include coding as well as configuration; depending on the service and its actual technology. Additionally to the implementation this phase also contains testing and staging activities. The tests should also be defined in the `Design` document (or referenced by it). The staging activities and environments need to be defined upon initialization of this process model. Furthermore, the release of the services and their components need to be defined in the initialization as well. The company should setup a release model and monitoring system in order to identify release problems and to avoid release ripple effects[1].

[1] See thesis from Bastian Schwartz for further details [Sch12].

We recommend to include at least the following stages (see section B.2.3 in the appendix for more details on the test levels):

Development Stage The development stage contains the development environments of the developers. Therefore, it typically does not contain all the data providers that provide data to the actual EMI. Hence, tests on this stage can only cover module tests, component integration tests, and service integration tests. If these tests are passed successfully then the service can be staged to the test stage.

Test Stage The test stage contains dedicated servers for the EMI services. However, it only needs to provide a similar infrastructure to the actual production stage. The technologies on this stage may differ (for example database technologies or messaging infrastructures). Furthermore, the stage may only contain stubs for the actual data providers of the EMI. However, this enables to run application integration tests on this stage. If they are passed successfully then the service(s) can be staged to the pre-production stage.

Pre-Production Stage The pre-production stage should contain a similar environment to the actual production stage. This requires to setup all technologies similar to those on the production stage and also requires to setup all the data providers similar to the actual production stage. These data providers, however, need to be controlled in order to setup test data. Thus, the pre-production stage allows to run EMI system tests which test the complete interaction between all services over the complete EMI. If these tests are passed successfully then the services may be released and staged to production.

Production Stage This provides the actual EMI that the metric customers use to satisfy their information needs.

The activities in the phase can be executed several times (iteratively) to create dedicated increments (typically one service); they may also be executed in parallel. Parallel execution can be more risky. Thus, the risks should be accessed before starting parallel construction activities. The `Design` document, however, should guide each of the parallel construction activities without adding too much additional risks.

The construction phase will still raise some issues even though the requirements analysis and prototyping phase before was performed thorough enough! However, these issues should just be minor changes thanks to the thorough inspections and evaluations performed before. These minor issues should, hence, be fixable during the construction phase and should not require to aboard the phase. Nevertheless, if mayor design flaws are found or large problems raise during the evaluation of the services with the metric customers in the later release stages then the construction phase needs to be aboarded. This will either spawn a redesign or will require a complete re-conception of the actual metric system. However, this is very unlikely to happen!

This already concludes our discussion on the construction phase of our process model. The following section will provide additional details to the most crucial phase for the long time success of the just constructed EMI: the operation phase.

12.2. The Operation Phase

The previous section briefly discussed the core activities in the construction phase. This section will provide additional information to the most central activities in the operation phase. The following section 12.2.1 will provide additional information on how to deploy and setup a new metric kernel. Deploying and setting up data adapters and visualization frontends is rather simple and does not require further documentation. Most importantly in this chapter we provide a list of best practices to handle common errors and exceptions in an EMI in section 12.2.2. We also provide some additional information about triggering a new iteration from the operation phase in the last section 12.2.3 of this chapter.

12.2.1. Deploy and Setup a new Metric Kernel

This section describes the setup of a metric kernel that is either new or was updated. The procedure is the same in both cases, hence, we only address the case "new metric kernel" in this section.

The database of a new metric kernel will be empty. Thus, it will not provide the indicators that the metric customer needs to answer her questions. Therefore, the new metric kernel will be in out-of-sync state after deployment. In order to resolve this, the metric kernel needs to receive all its relevant measurement messages to fill the database. The data could be resent from the data provider via the data adapter but this would create a large load in the data provider. Besides, the resulting measurement messages are already stored in the message cache. Thus, the operator simply needs to utilize the `Resend API` in the message cache to resend all relevant measurement messages. The new metric kernel will receive all these messages and setup its database accordingly. Therefore, after the messages are resent the metric kernel is in-sync with its data sources.

The additional messages will also be received by all other metric kernels in the EMI which would potentially create a large server load on the nodes of the metric kernels and their databases. However, the metric kernels should be robust against receiving the same message multiple times (see section 7.2.4). Therefore, most existing metric kernels will simply ignore the messages. Hence, this will not create a large load on the servers.

12.2.2. Best Practices for Handling Common Errors and Exceptions

The following subsections contain typical errors and exceptions that can occur when operating an EMI. Most of these errors and exceptions discussed are due to the breakdown of one of the components of the EMI. Therefore, for each of these problems we provide a way to identify the problem and steps to follow in order to resolve the problem and repair potential damage. We start each subsection with a short description of the problem and its implications.

Failure of a Data Adapter

The failure of a data adapter will potentially result in data loss in the EMI. Data adapters following the Push-Forward adapter pattern (see section 5.4) are most likely to miss some data that is pushed towards them while they are offline. The recovery steps depend on the adapter pattern of the data adapter. Therefore we will differentiate the recovery steps for each of the adapter pattern.

Detection The failure of a data adapter can be detected via the monitoring system. The data adapter will be in offline or unknown operation state. Furthermore, the metric customers may complain due to missing data in the EMI.

Recovery Steps – Pull-Forward Adapter Pull forward type data adapters are triggered by an internal timer which triggers the data adaption periodically. When the data adapter breaks down this timer needs to be bypassed to get all the data for the timeframe in which the adapter was not available.

1. Fix problem in data adapter.
2. Redeploy the data adapter in maintenance state (no periodical triggering).
3. Trigger the pull method to get all the data from the data provider from the missing time frame.
4. Reinitiate the periodical trigger and set operation state to online.

Recovery Steps – Invoke-Pull Adapter Similar to the pull forward type data adapters the data adaption from invoke pull type data adapters is also triggered. However, the trigger is not periodical but other messages on the EMDB. These messages are cached in the message cache. Therefore, when an invoke pull typed data adapter breaks down it is sufficient to simply resend all the messages that where send during the down time of the data adapter. These messages will then retrigger the data adaption without any additional changes to the data adapter.

1. Fix problem in data adapter
2. Redeploy the data adapter in online state
3. Resend all messages for the downtime of the data adapter (maybe filtered to the triggering messages of the data adapter) via the resend-API of the message cache.

Recovery Steps – Invoke-Dump Adapter Invoke dump type data adapters, by their nature, always adapt the full data from the data adapter. Therefore, when such a data adapter breaks down it is sufficient to simply retrigger the dump because it will adapt all (missing) data from the data adapter. The recovery steps are similar to the invoke-pull adapter pattern above but step 3 is replaced by:

3. Retrigger dump on the data adapter (for example via a dedicated command message)

Data Loss Prevention – Push-Forward Adapter The main problem with push forward style data adapters is their inversion-of-control style design. They receive data from the data provider whenever the data in the data provider changes and they only receive the changed data! Thus, if such a data adapter breaks down then the call from the data provider and its containing data is lost. Therefore, we focus on methods to prevent the data loss situation rather then providing recovery steps. Multiple alternatives exist to prevent the data loss. The two most popular once that we used in our field studies are:

Multiple deployed data adapters + Load Balancer The most trivial way to prevent data loss for push forward styled data adapters is to deploy multiple instances of the data adapter on different nodes in the EMI. All the data adapters connect to the same EMDB. Located in front of these multiple data adapters (from a data flow perspective) a load balancer simply delegates the call from the data provider to a working version of one of the data adapters. The load balancer should, thus, be coupled with the monitoring system. If one of the data adapters fails then one of the other running data adapters will immediately take over and data loss is prevented. However, the load balancer then becomes a single point of failure and data is potentially lost if the load balancer fails. However, load balancers are build to be very robust and almost never fail. Additionally, there exist additional methods to increase the robustness of load balancers to ensure 24/7 operation. However, this design increases the maintenance effort when updating the data adapters and the complexity of the EMI and the staging environments.

(Caching) Proxy between data provider and data adapter Another alternative to prevent data loss to push forward style data adapters is to use one (or several) proxies[2] in front of the data adapter that are able to cache the requests. The proxy will echo the actual APIs of the data adapter and be used by the plugins in the data provider. The data from the data provider is then transfered to the proxy which checks the monitoring service to get the availability of the data adapter. If the data adapter is not online the proxy will cache the data. When the data adapter gets back online again all the missing data can be retrieved from the cache and be played back to the data adapter. This is a similar behavior to the recovery of invoke-pull adapter and failures in metric kernels. However, this creates a single point of failure at the proxy and potential performance problems when all requests need to be transfered through one proxy. This can be solved by deploying multiple proxies that are located behind a load balancer which distributes the calls; similar to above. Furthermore, this will also increase the complexity of the EMI and the staging environment. However, it will most likely not increase maintenance effort because the data adapter is only deployed once.

[2] Additional information on such a proxy, technical details, and test cases for the proxy can be found in the bachelor thesis of Gordon Lawrenz [Law14].

Failure of a Metric Kernel

The failure of a metric kernel will result in missing indicators for monitors in visualization frontends. Thus, metric customers are unable to answer questions related to these indicators. Furthermore, the metric kernel will *miss* measurement messages on the EMDB during its down-time. Hence, when redeploying the metric kernel it will most likely be in out-of-sync state. Therefore the operator needs to perform similar actions as to deploying a new metric kernel to get it back in-sync again.

Detection The failure of a data adapter can be detected via the monitoring system. The metric kernel will be in offline or unknown operation state. Furthermore, the metric customers may complain due to missing data in certain monitors that are feed by the particular metric kernel.

Recovery Steps Similar to above the operator needs to resend the missing messages that occurred during the down-time of the metric kernel via the message cache.

1. Fix problem in the metric kernel
2. Redeploy the metric kernel in maintenance state
3. Resend all (or filtered) messages using the `Resend API` from the message cache
4. Set the metric kernel to online state when it processed all messages and gets back in-sync

Failure of a Visualization Frontend

The failure of a visualization frontend does not result in any problems with the reception and processing of measurement messages in the core components. Therefore, a failure in a visualization frontend or another service in the visualization layer does not require any additional steps other then fixing and redeploying the defective service. This, isolation of failures which only temporarily disable certain components, again, shows the advantages of the design of the reference architecture.

Failure of the EMDB Messaging System

The enterprise measurement data bus (EMDB) and the underlying messaging system provide the backbone of an EMI by transporting its measurement messages. Thus, a failure in this part can potentially result in data loss because the message cache also does not receive any messages. Several solutions exist to prevent or recover from this failure. We will discuss the two most prominent solutions in the following steps.

Detection The detection of failure of the EMDB is not trivial because all operation system are not working either. Maybe the failure will produce according log messages in the services. However, because the messaging system is not working these messages do not get delivered to the central logging service. Therefore, a good

indicator for a failure of the messaging system is the monitoring service reporting unknown service state for all services in the EMI (including itself).

Variant 1 – Multiple Messaging Systems (Prevention) Failures in the messaging system can be reduced if multiple systems plus automatic failover is provided. If one of the systems fail then another one will automatically take over. However this is hard to configure, operate, and maintain because automatic failover is not provided out-of-the box for most messaging systems. Also these multiple systems need to be setup in the pre-production stage to simulate and test the failover and failure prevention.

Variant 2 – Local Message Caching (Recovery) In order to avoid multiple messaging system the EMI development platform, which provides the message senders, can also include local message caches. These local caches are then able to cache all messages that could not be send, due to a failure of the messaging system, in the service. When the messaging system comes back online a centralized message over the command topic can force the local caches to flush their messages to the EMDB. This mechanism prevents data loss as long as none of the services breakdown and the local caching is working. However, this solution requires additional construction and maintenance effort for the EMI development platform and a service to trigger the flush of the local caches. Nevertheless, the local caches can also be used to check messages in a test without actually sending the messages to the EMDB.

1. Fix problem in the messaging system
2. Restart the messaging system
3. Force flushing of the local message caches of the services

Failure of the EMDB Message Cache

The message cache is the most important system to recover in case of service failure. Therefore, the message cache needs to be operational in order to prevent data loss in case of a service failure. Again, similar to a failure of the EMDB messaging system we present two different variants to prevent or recover from a (complete) message cache failure.

Detection The failure of the message cache can be detected via the monitoring system.

Variant 1 – Multiple Message Caches (Prevention) Similar to preventing a failure of the messaging system of the EMDB, multiple instances of the message cache can be deployed each running on different nodes using different databases. If one of the caches fails then another cache will still get all messages on the EMDB and the failed cache can be resetup with the data from the running caches. Similar to above, this variant produces a lot of configuration, operation, and maintenance effort. Furthermore, it also requires additional server resources to ensure that the messages caches are running on independent server nodes.

Variant 2 – Local Message Caching (Recovery) Similar to above, a local cache in each service could detect the failure of the message cache and cache each message until the message cache comes back online. When the message cache becomes operational again then all local caches can be dumped and the resulting messages will be cached in the central message cache. Similar to above this required additional effort for the construction and maintenance of the EMI development platform.

Failure in the Monitoring System

A failure in the monitoring system will prevent the operator from detecting failures in other services in the EMI. Therefore, the monitoring system needs to be fixed and redeployed as fast as possible. Furthermore, the EUrEKA consumers can no longer check the service state of the metric kernels. Hence, they will assume that the metric kernel is online which may result in wrong data or timeout errors if the kernel is currently being maintained or offline. All other operations in the EMI will work normally.

Failure in the Logging System

The central logging system provides the metric experts, the operators, and the developers with aggregated log information from all services in an EMI. A failure in this system will prevent this and will potentially result in missing log data in the logging service. However, typically the services will also write log data into local logs on their servers. Hence, no data is lost. Nevertheless, the failure in the logging system should be fixed and the system should be redeployed as fast as possible.

Failure in the Directory System

The directory service provides services in an EMI with a mechanism to resolve synonyms in various directories. Hence, the synonyms will not be resolved if the directory system fails. This may lead to wrong indicators because data is not assigned correctly in the metric kernels. The recovery steps depend on the location at which the synonym resolving is implemented in the metric kernels (see section 5.5.2). Hence, we will address different recovery variates in the following brief description.

Detection A failure of the directory system can be detected via the monitoring system. Furthermore, the metric customers may complain about wrong or missing values. Additionally, the services may log information regarding timeouts when trying to connect to the directory service.

Variant 1 – Synonym Resolving at Indicator Access When the synonyms can be resolved only at the indicator access then the data storage mechanisms are not influenced by a failure of the directory system. Hence, it is sufficient to redeploy or restart the directory system and ensure that everything is working again. No further steps are required.

Variant 2 – Synonym Resolving on Message Reception Synonym resolving at indicator access can complicate the database queries and the calculation mechanism. Hence, resolving the synonyms on message reception before storing the data in the database of the kernel was the more common option in our field studies. However, this requires additional effort when the directory system fails because a failure will result in wrong data in the database of the metric kernel. Therefore, when the directory system is online again the operator needs to resend all messages that occurred during the downtime of the directory system. This will result in correct data in the database of the metric kernel. However, the database may still store relics of calculations for the synonyms. If these disturb the calculation or bother the metric experts or customers then the operator needs to manually remove them from the database.

1. Fix problem in the directory system
2. Restart the directory system
3. Resend all messages for the downtime of the directory system via the resend-API of the message cache
4. Optionally: tidy up the database from the calculation relics caused by not resolving synonyms

12.2.3. Triggering a new Iteration

We already discussed the monitoring information needs of metric experts in the discussion of the monitoring system in section 6.1. Most importantly the metric experts require analytical data for key performance indicators from the usage of the different services in an EMI. They can then use this information to assess the quality of the metric portfolio which can lead to the identification of "holes" (uncovered information needs from metric customers) and unused metrics and visualizations. With this information the metric expert can initiate a new iteration of the process. The information will then guide the activities in the conception phase; most importantly the **`Plan RE`** subactivity in the **`Requirements Gathering`** activity.

12.3. Summary

This chapter briefly discussed the construction and the operation phase of our engineering process model. Most importantly for the construction phase we discussed the minimal release stages: development, test, pre-production, and production. These stages support the different levels of tests for EMI services and the integration between these services presented in the appendix in section B.2.3. Additional activities and specific artifacts for the construction phase heavily depend on the existing tools, technologies, and processes of the company. Therefore, they need to be defined when instantiating the process model.

The discussion on the operation phase, most importantly, presented various best practices on how to handle various failures in an EMI. Most importantly, this showed the importance of certain services in an EMI like the message cache in the EMDB and the monitoring system. They both play a crucial role in the detection of failures and recovery from them. We also discussed data loss prevention for push-forward style data adapter and message cache failure. Data loss should, obviously, be avoided in an EMI because it will inevitably lead to wrong calculation results. Therefore, it is crucial to also consider these failure scenarios when designing EMI services, which is the reason why we specifically included a failure section in the `Design` document (see section B.3 in the appendix).

This concludes the definition and discussion of our engineering process model in our metric systems engineering approach MeDIC. The following part IV will present the evaluation of our process model and our reference architecture by selected field studies. Furthermore, we will also provide a brief introduction to the tools and framework, which we build to support the field studies.

Part IV.

Evaluation, Tool Support, and Lessons Learned

13
Evaluation by Selected Field Studies

Evaluating an engineering approach is a hard task! The benefits and weaknesses are not easily found in experiments and case studies due to their complexity and interconnections between the different activities, artifacts, and technical solutions [Woh12]. Furthermore, our approach aims to support the engineering of metric systems in industrial environments. Thus, an evaluation needs to be performed in this environment in order to provide feasible results. However, an evaluation in an industrial environment can only be performed as a dedicated project. Like all other projects, these projects always face a lot of constraints. Most importantly an evaluation in an industrial environment needs to provide some (short term) benefit for the cooperation partners. Otherwise, they will most likely not provide the necessary (expensive and rare) resources to perform the evaluation[1]. This makes it very hard to perform case studies and controlled experiments in industrial environments.

Our evaluation is centered around a number of field studies, which aim on answering research question Q5 (see section 1.3). Furthermore, by our evaluation approach we provide examples for the application of the reference architecture as well as the utilization of the process model. These field studies where performed as projects which provide a usable metric system with a production-ready metric infrastructure for our cooperation partners. Therefore, the field studies always utilized our process model and instantiated (parts of) our reference architecture. However, we tried to keep the focus of the field studies as narrow as possible. Thus, some of the studies emphasized the process evaluation whereas some others emphasized the evaluation of the architecture and technical aspects. Each field study description in the following sections starts with the detailed research question of the field study. The specific research questions are sub-question to our initial research question Q5 (see section 1.3). We also provide a header with additional details to the field study. It again lists the focus and the environment of the field study as well as some key metrics for their size.

Each work and thesis performed within the context of this research project was evaluated individually using different approaches. Hence, we have over 40 individual evaluation for various parts of our engineering approach at hand. However, they only evaluate very small parts of the approach, which we do not want to discuss out of context[2]. Thus, we will not rediscuss all these different very detailed evaluations and focus on three core field studies in this thesis.

[1] We also faced this problem in our cooperation projects. In one case it resulted in aborting a project!
[2] Chapter C in the appendix lists all the theses for further studies on the detailed evaluations

We used various methods to evaluate all the individual aspects of our approach. For example, most of the GUIs and user focused tool projects were evaluated using a combination of field studies and additional questionnaire supported interviews. The questionnaires often implemented the ISO 9241 standard in order to provide a common evaluation basis. We also used constructive interaction [ODR84] in order to evaluate some of the user interfaces. Most of the work used the ISO 9126 / ISO 25000 quality models in order to perform qualitative evaluations based on the different quality criteria of the models.

Our evaluation in this thesis is based on three selected field studies. We start with a long time field study that we conducted with a large IT service provider for an insurance company in section 13.1. Contrasting this study, section 13.2 presents a field study in an academic environment which should provide a very generic metric system for the sse lab infrastructure. We already published some of the aspects mentioned in these two field studies in two articles [VL14, VLS14]. However, we added additional details to the processes and the architecture. The last field study in section 13.3 presents our experience with using our reference architecture and process model to build a tool to analyze ticket flows. The corresponding metric system was also evaluated with two large IT service providers.

Each section provides a short introduction of the environment and the main requirements for the specific metric system. For each field study we then present additional details to the specific engineering process utilized in the specific field study which was instantiated from our process model. Afterwards we provide some details on the design of the EMI that implemented the required metric portfolio. We conclude each field study with a condensed list of experiences and a small summary.

13.1. Project Risk Metric System for a Large IT Service Provider

Research Question (Q5.1):

> Are the MeDIC process model and MeDIC reference architecture applicable in a large industrial environment?

Focus:	Process model (especially requirements analysis and design)
Environment:	Large IT service provider
Code size: Project specific	$<$ 10k LOC
Code size: Overall incl. EMI Framework	$<$ 40k LOC
Overall Effort:	$\approx$ 64 PD (+ one partial diploma thesis)

This field study was conducted in cooperation with a large IT service provider for an insurance company. Thus they need to deal with the legacy of very old systems as well

as the challenges from modern service-based infrastructures. Hence, the project sizes vary across a large scale; some development projects can be small (about 100 person days) others are very large (up to 35,000 person days). This IT provider is CMMI Level 3 certified. Therefore, all development projects need to apply the organization's standard development process.

The goal of this field study was to engineer a flexible metric system for the project managers. We worked together with the engineering process group and were supported by two metric experts from within the company. In the later stages of the field study the company started a dedicated project to support the development and pre-production stages of the metric infrastructure including development and operation teams.

This field study was initiated because of problems[3] in the usage of metric based monitoring dashboards for project managers. We noticed that the information needs of the project managers were no longer aligned with the needs answered in the dashboard template provided by the company wide process. Hence, we initiated our engineering process in order to systematically address the issues. The following subsection provides additional details to the instantiation of the process model.

13.1.1. Process

Figure 13.1 depicts the main steps of the requirements phase and the two construction increments of the metric system. The three main requirements sub-processes are located on the left. They were executed iteratively (indicated by the loop-icon). The whole requirements process could be executed iteratively as well if excessive flaws are detected during the **`Prototype and Evaluate`** or the **`Plan Increments`** subprocess.

The **`requirements gathering`** subprocesses started with an assessment of existing metric systems used by the IT service provider. This analysis revealed that the information needs of the project managers were changing. Thus, we conducted interviews to systematically gather these changes [VLJ13]. We then analyzed the changed information needs and developed a prototype for a new metric-based monitoring dashboard which was evaluated by the project managers. Furthermore, we developed monitor prototypes for the monitors on the dashboard. It took some iterations to pin-point the core interactions and needs. Then we analyzed all gathered requirements and specified the realization increments. The first one focused on metrics to analyze project risks, the second focused on metrics based on error and enhancement tickets.

Risk Metrics Increment

We further enhanced and specialized the monitor prototypes focusing on the visualizations and diagrams that we wanted to address. After just a few iterations we were able to provide the central visualizations and diagrams required for the project managers. Two of these final diagrams are shown in figure 13.2. The first is a classic Cartesian chart with four bar and one line chart showing the number of risks in specific states and the

[3] See [VLJ13] for additional details.

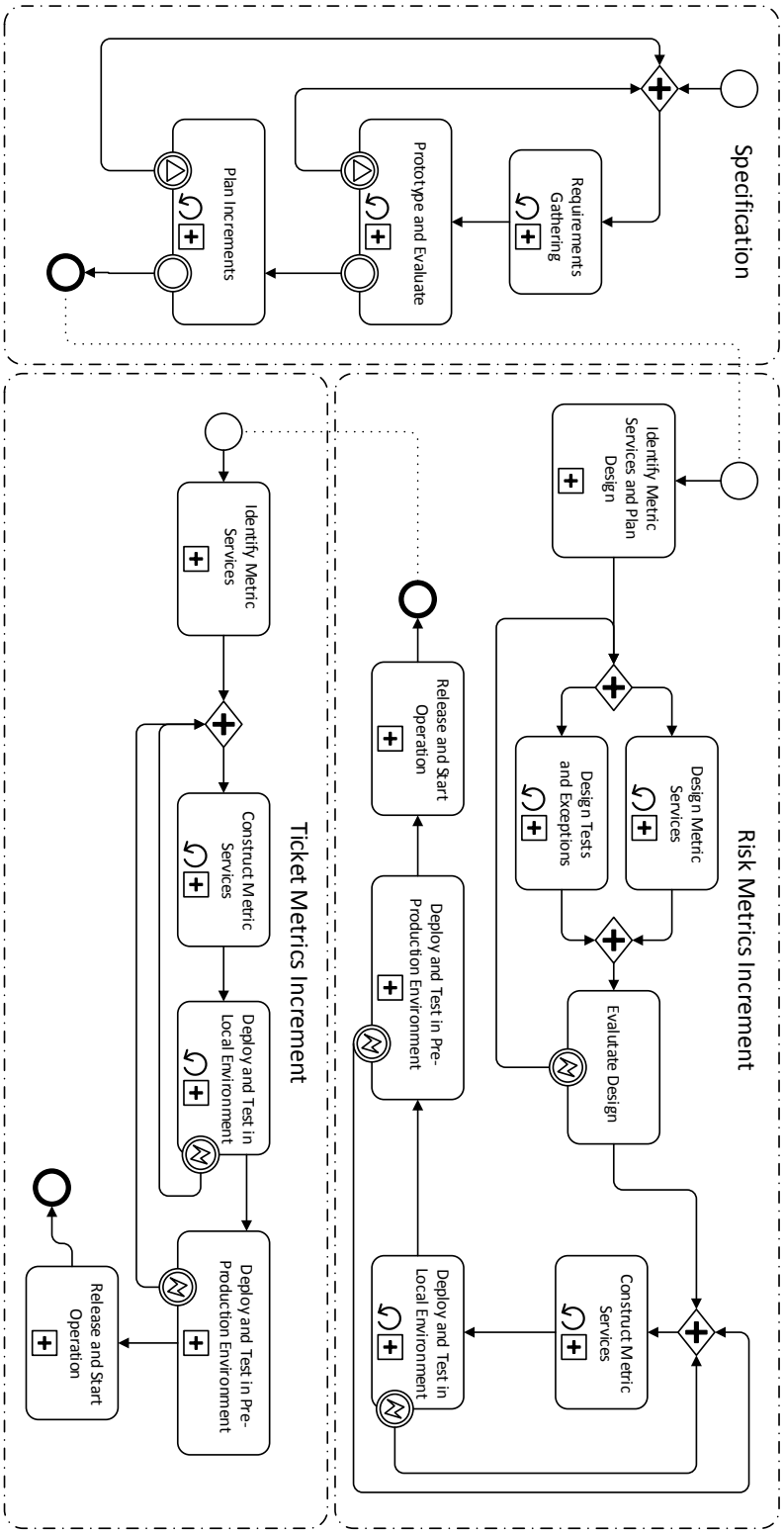


Figure 13.1.: BPMN diagram of the main process steps used to develop the metric system at the IT service provider

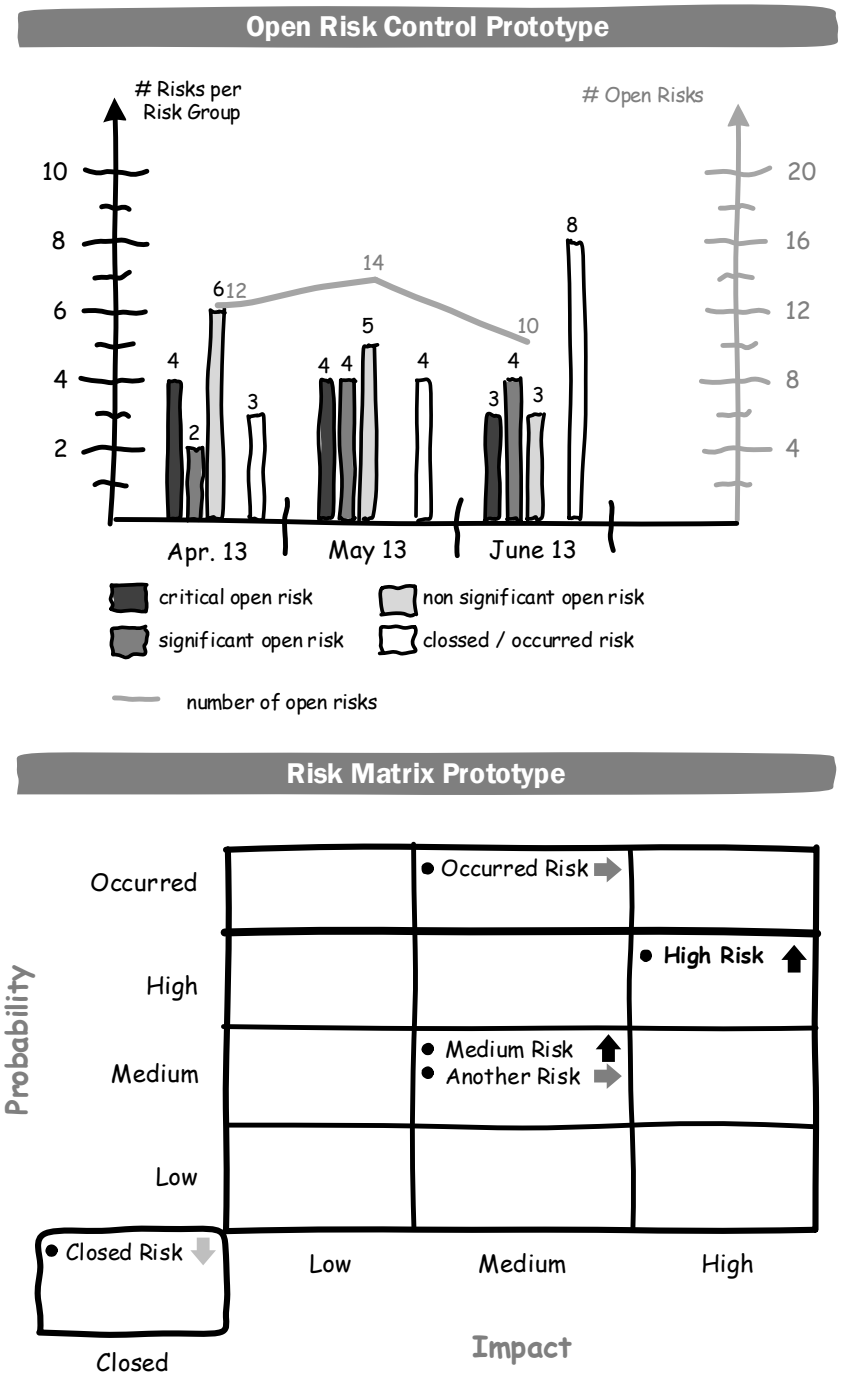


Figure 13.2.: Prototypes for metric-based risk monitors: Risk Matrix and Open Risk Control

overall number of open risks each on a monthly basis. The second one is an enhancement of a traditional risk matrix. It shows the impact on the bottom and the probability on the left, each on a three item scale from low to high. The top row contains the risks which did occur and the lower left square contains closed risks. In addition to this we added an icon after each risk to indicate how it changed[4]. These changes are tracked on a monthly timing as well.

Project risks were documented using dedicated Excel sheets. These sheets are based on a template which is part of the standard development process of the company and mandatory for the projects. During risk workshops these risk sheets are filled with new risks and existing risks get updated. Additionally, project managers update the risks if something that influences the risk changes (for example a counter measure for the risk is showing to be effective). The risk Excel sheets are stored in CVS repositories.

Handling risks in an Excel sheet is a risk on its own due to lack of consistence checks, constraints, and solid work-flow modeling. Hence, it was planned to use Atlassian Jira to model and store risks in the future. Unfortunately, Jira does not understand the semantic of a risk and hence is not able to provide the risk matrix or bar charts mentioned above. This additionally enforced the need for an independent visualization of risks.

We started the increment by identifying and planning the design of the required metric services (data adapters, metric kernels, visualizations, and dashboards). The design was again performed iteratively. We first created a rough version of the EMI and specifically focused on the integration part. Then, we started the detailed design of the required data adapters and metric kernels. In parallel we conducted several workshops to discuss possible failures and exception behavior in order to define meaningful test scenarios and test cases. We evaluated the design and did some minor changes after the first iteration.

We then started the construction of the metric services. The integration of the newly constructed services worked flawless. We believe a reason for the success was the good and thorough design before we started construction. Every service could be tested in a local EMI environment. We also continuously deployed the current versions of the metric services to a pre-production environment. This enabled continuous testing by the metric experts and provided important feedback to the developers. This construction of the metric services was finished after roughly 1.5 months and we were able to release a first version and start pre-production tests with metric customers.

Ticket Metrics Increment

The focus of the second increment was to implement a dashboard for monitoring ticket-based metrics. It was also started by identifying the required metric services. It was planned that the company performed this increment by themselves. Unfortunately, due to resource constraints (missing architects) and project management decisions the design activities were skipped. Hence, the developer had to perform the design "on the fly". The local development of the metric services again worked smoothly due to

[4] Also see the example in section 5.3.3, which provides an example for a metric kernel description that could feed such a risk matrix.

the well designed reference architecture and development and operation tool support. The deployment and test in the pre-production environment, however, did not work as flawlessly as anticipated. The reasons were problems with the deployment configuration and incompatible interfaces as well as configuration issues within the dashboard services. The development also took longer than anticipated. A lot of these problems, we believe, are due to the missing design phase, lack of design experience, and lacking design and integration support as well as arguable decisions in the management of the project. Hence, this also mirrors the outcome and experience with traditional construction of measurement infrastructures that lack the support of our engineering process model. Due to these problems we will focus on the risk increment in the following architecture section.

13.1.2. Risk Metrics Architecture

This section presents a brief summary of the design of the EMI for the risk metrics. Figure 13.3 depicts an overview of the static architecture of the developed EMI for the risk metrics. Contrasting our view on the technical reference architecture in section 5 the view is turned 90° starting with the data providers at the left hand side. From there we show the measurement layer with three data adapters (two push-forward, one invoke-push), the data transport layer with the typical three EMI topics (see section 5.2), and the calculation and storage layer with the the risk metric kernel on the right. The following paragraphs provide additional details to the data adaption and calculation.

Figure 13.4 provides a UML activity diagram with additional details to the adaption of the risks and the interaction between the different elements: CVS server, commit event REST gateway, and Excel-list adapter. As described above, the risk Excel sheets are stored in a central CVS repository. We used a small script in a commit hook of the version control system that calls the commit event gateway in the EMI on a change of a file in order to notify the EMI of a file change. The Excel list data adapter implements the invoke-push adapter pattern using this commit.cvs event. The event just contains information about the file that was changed but not the file itself. Thus, the Excel-list adapter accesses a viewvc server via http to get the specific revision of the Excel file. The data adapter then feeds the Excel file to each strategy that is configured to accept the specific sheet. The risk list adapter strategy analyses the risk list and sends the result, a risk list message, via the base bus of the EMI. An alternative would be an exclusive data adapter just for the risk list. We decided to implement a generic adapter with specific strategies because we anticipated to adapt more Excel sheets in the near future using the same mechanism.

For Jira we again created a small plugin that is executed whenever a risk ticket changes. This plugin then calls the REST API of the risk gateway in the EMI which again creates a risk message on the base bus. This is just a textbook instantiation of a push-forward style data adapter that generates risk messages based on the input of its REST API. Thus, we do not provide additional information to this (see section 5.4 for further details).

The risk messages on the base bus are received by the risk metric kernel. Figure 13.5 provides a UML activity diagram of all the steps performed in the risk kernel between reception and storage. The kernel first checks consistency constrains of the incoming risks

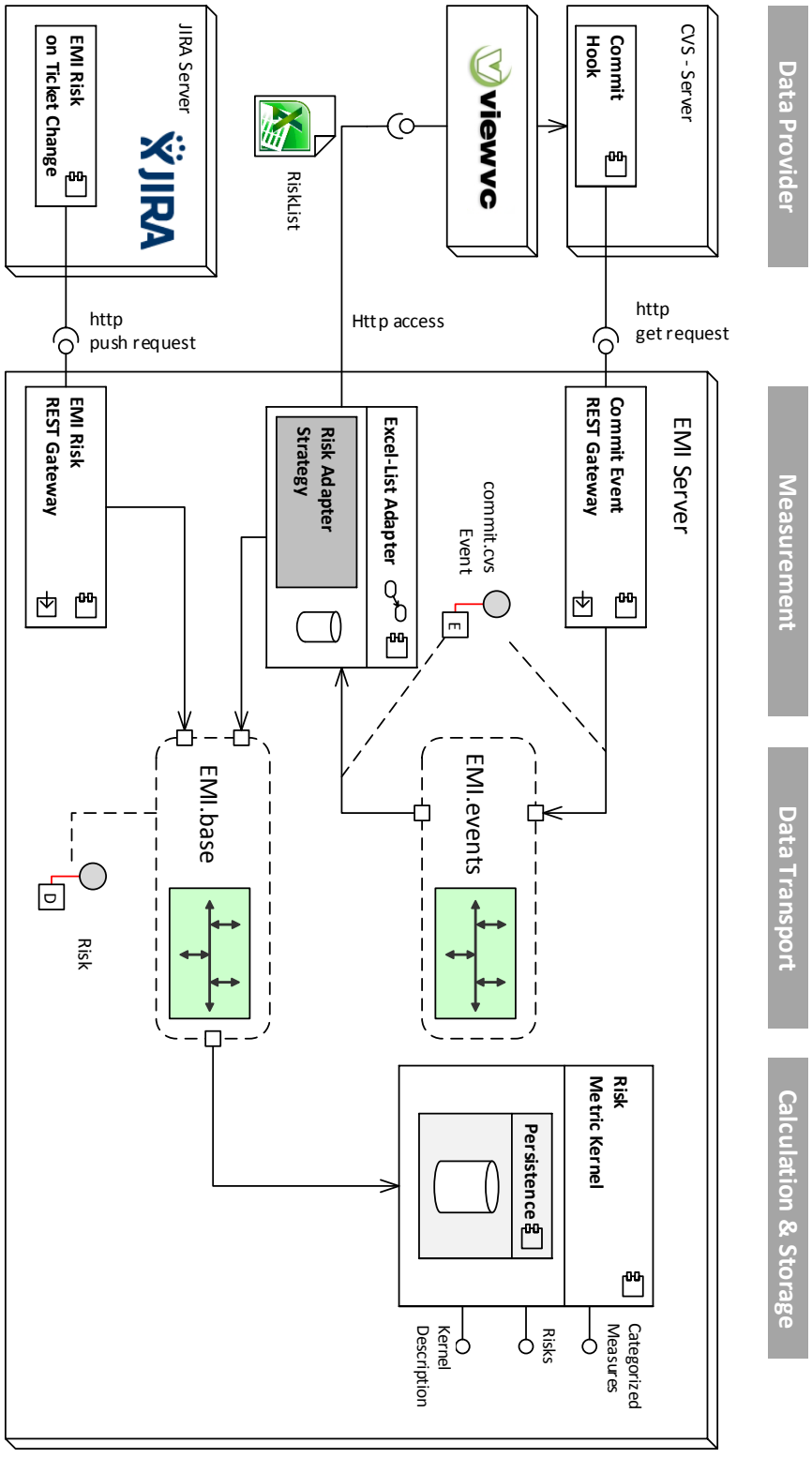


Figure 13.3.: Static architecture overview of the data adapter and metric kernel of the EMI for risk metrics at our cooperation partner.

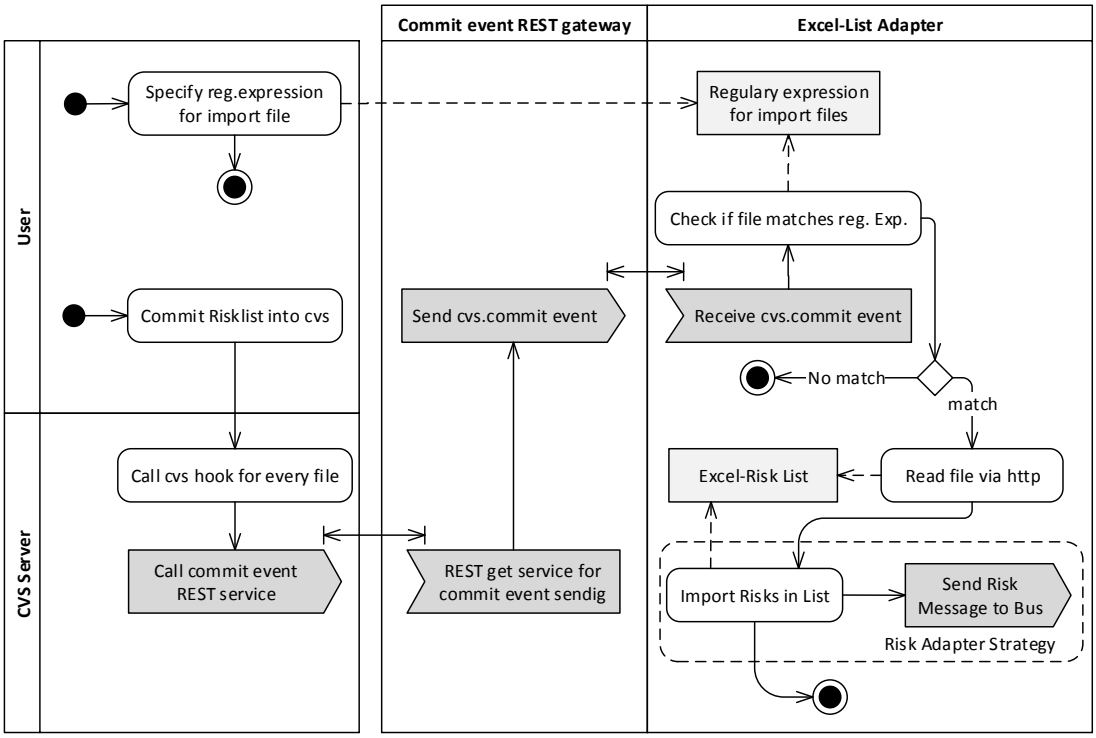


Figure 13.4.: UML activity diagram for the data adaption for risk metrics.

against the stored risks due to possible data corruption in the Excel sheets (duplicated ids, deleted risks, etc.). Some of these errors can be corrected by the kernel; some of them result in a rejection of the in-coming message and error messages to the central logging service as well as indications in the monitoring system. If the risk message passes the check the risk is stored in the data base of the metric kernel. The metrics are calculated on request via the REST APIs of the risk metric kernel.

The data model of the risk kernel as UML class diagram is provided in figure 13.6. This data model does not reflect the structure of the messages but is tuned to provide fast and easy calculation of the required metrics. We used this data model to feed our Gargoyle code generator[5] to generate the data abstraction layer and data abstraction objects, which abstract the database. All these metrics are defined on a monthly timescale. Therefore, all the important (dynamic) information about a risk is stored in the `RiskHistory` table. The primary key of the data is a combination from the risk, the eom, and the report month. Therefore, most of the metrics are simply calculated by counting the number of specifically filtered history entries.

[5]See the thesis of Tobias Löwenthal, Steffen Conrad, Tristan Langer, Claude Mangen, and Michael Krain for additional details on the Gargoyle code generator [Löw11, Con12, Man12, Kre12, Lan12].

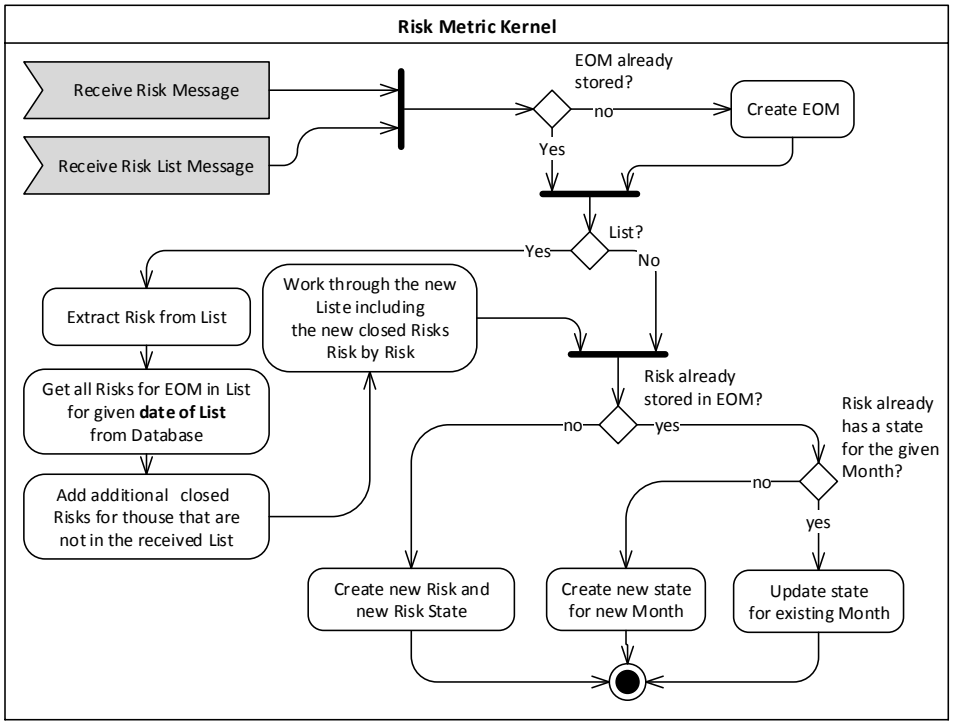


Figure 13.5.: UML activity diagram for the data storage and pre-calculation of the risk metric kernel.

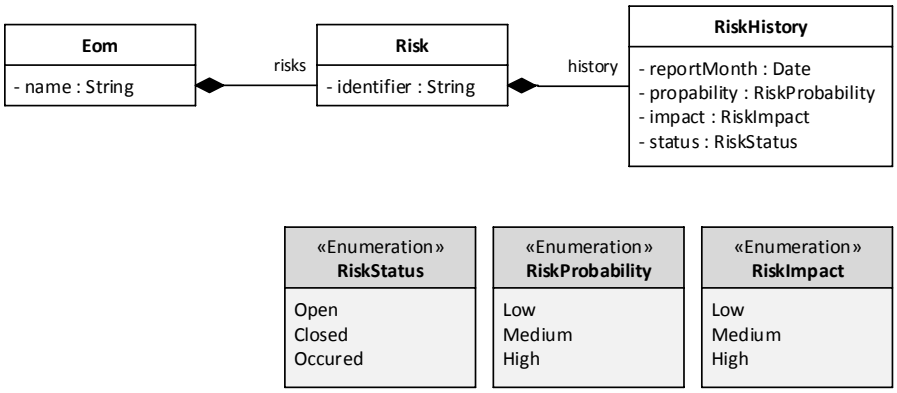


Figure 13.6.: UML class diagram for the data model of the risk metric kernel.

13.1.3. Experience and Best Practices

We gained a lot of experience on using the reference architecture and our engineering process model in this field study. Most importantly, we used our engineering process model to find and evaluate the information needs of the metric customers using prototypes. We also instantiated the design and construction phases from our process model for the risk

metric increment. The resulting EMI was an instantiation of our reference architecture. It contained different types of data adapters and a metric kernel that followed our design guides. The core experiences from this field study are:

The reference architecture proved to be successful! The CMMI level 3 certified organization required an audit of the reference architecture before we used it inside the organization. The core goal of this is to check the compliance against existing architecture rules and guidelines. The EMI reference architecture was reviewed by several architects and then feed to the architecture management board. It was very well received by all the reviewers as well as the board which lead to the reference architecture passing the audit.

Push-Invoke pattern successfully adapts Excel sheets! Initially we were struggling with the decision on how to adapt the Excel-based risk lists. We thought about developing dedicated Excel plugins to synchronize the lists via the click of a button. This would require additional clicks when working with risk lists though and we believe that this would lead to some troubles later on[Joh01]. We looked for an alternative and ended up using our invoke push adaption mechanism on the risk lists stored in the CVS version control system as described. This does not require additional attention from anyone working with the risk list and due to a very open and flexible design of the Excel list adapter it can easily be extended to adapt other types of Excel sheets.

Unstructured data requires additional attention! Adapting data from a database or other systems like Jira requires very little consistency checks because it is very hard to corrupt the data. Unstructured data like Excel sheets or CSV files, however, require a lot of attention in the metric kernel (and maybe the data adapters) because a lot can (and will!) go wrong. We did an extensive workshop session to discuss possible data corruption scenarios. For some of them we also defined recovery mechanisms (for example a missing row in an excel sheet which can be detected by the metric kernel).

Developing, designing and operating an EMI requires training and time! The EMI reference architecture provides a very good and structured overview of the different parts of an actual EMI. Using the reference architecture and developing and operating the specific metric services, however, requires time and training. During this project we trained two metric experts and a few developers and architects.

Involve metric experts from the beginning! We recommend that at least one metric expert participates in the interviews or workshops because metric experts become particularly useful if the discussion is to one-sided or stuck.

Apply best practices to identify metrics! We typically use GQM to analyze the information needs. However, we very rarely use the formalized goal definition because we experienced that it leads to unnecessary and narrow discussions. Furthermore, we try to align the metrics with the measurement information model of ISO 15939. We recommend to keep your set of metrics as simple and small as possible without sacrificing metrics for a dedicated information need.

Develop prototypes iteratively and incrementally! We recommend developing the monitors incrementally one after the other. Each monitor itself should be developed iteratively. After a new monitor is added the dashboard has to be evaluated to ensure that all monitors together cover the information needs addressed so far.

Always perform a design phase! All software engineering text books and guidelines emphasize on the importance of the design activities. Additionally, agile process models like scrum include design activities in the "backlog refinement". However, if the development team does not have a solid understanding of the reference architecture it is important to include the design phase before handing it to development and to evaluate the designs.

Provide tool support and frameworks! Using a dedicated development platform according to a reference architecture can drastically reduce the development effort and increase reference architecture compliance. It supports the development with dedicated hot-spots and pre-fabricated solutions for typical problems and can provide ready-to-use services as well.

13.2. Software Project Metrics System for SSE Lab

Research Question (Q5.2):

> Are the MeDIC process model and MeDIC reference architecture applicable in a university/small industrial environment?

Focus:	Process model and reference architecture
Environment:	University (research projects)
Code size: Project specific	$<$ 10k LOC
Code size: Overall incl. EMI Framework	$<$ 20k LOC
Overall Effort:	$\approx$ 23 PD (+ one bachelor thesis and a partial diploma thesis)

SSELab is a management infrastructure mainly supporting software development projects at RWTH Aachen University [HKR12]. It integrates key services like version control systems (git and SVN), wikis, and change request management systems (TRAC) into one coherent platform. Currently SSELab hosts over 700 projects. Our analysis showed a very heterogeneous project environment which is dominated by software development projects. However, SSELab is also used for the administration of organizational projects and scientific projects such as paper or thesis projects as well as teaching projects like lab courses. Even though SSELab offers a lot of features and functionality it lacks the support for measurements. Hence, the goal of this field study was to investigate what metrics are required for the project managers of SSELab projects and to develop a maintainable, robust, and flexible measurement infrastructure that enables the calculation and measurement of the metrics in SSELab.

13.2.1. Process

The main goal was to engineer a metric-based monitoring dashboard template for project managers. We conducted interviews as requirements gathering technique to get a broad feedback on the information needs of different project managers. Following our process model, we used a questionnaire to keep the project managers focused on the important aspects. However, some of the most interesting information needs were not directly related to our questions.

Afterwards, the results of the interviews were integrated into a large mind map following the GQM principle. From this we derived key information needs and their corresponding metric-based monitors to be included in the dashboard prototype depicted in figure 13.7. This prototype was designed following the dashboard design principles proposed by Few with the most important visualizations (Bullet Graphs) on the top left [Few06, Few12]. The dashboard features visualization of source code metrics and statistical metrics on version control activities (e.g. number of commits per week) and issue tracking (e.g.

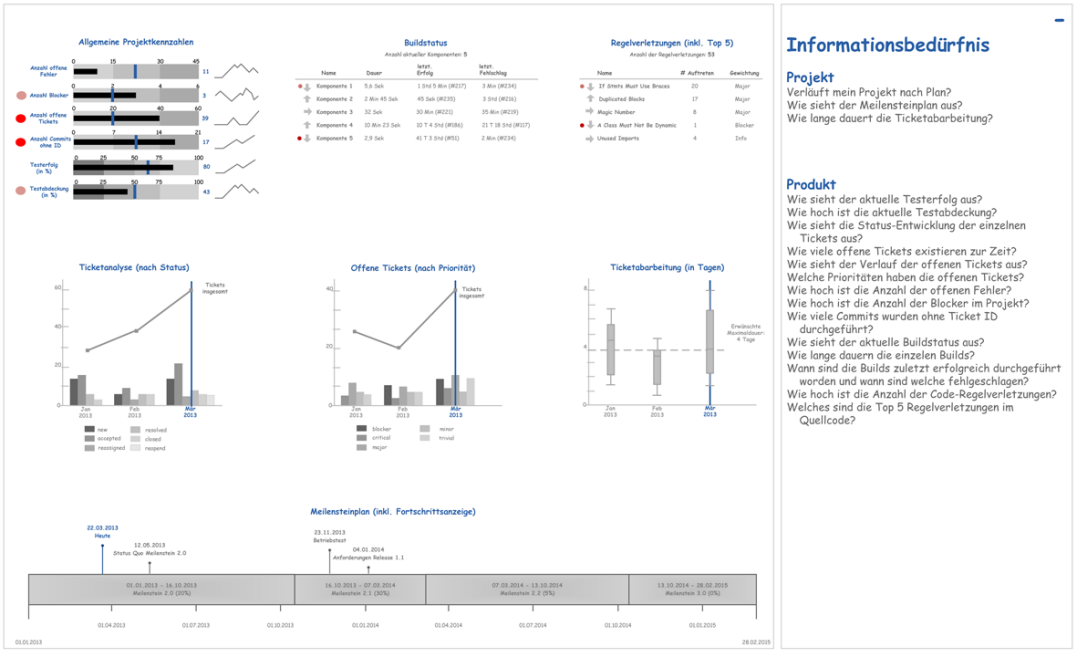


Figure 13.7.: Prototype for the specific SSE Lab metric-based monitoring dashboard.

number of open and closed issues per week). This dashboard prototype was then sent to the project managers via mail for evaluation. We received some feedback and then conducted additional interviews with selected project managers.

Based on the feedback we changed the interaction and updated some of the metrics and visualizations. The modified prototypes were again evaluated by the project managers without any major findings. We then started the design phase. The design phase, again, followed our process model very closely and we defined the various aspects required in the `Design` document. Most importantly we defined the architecture of the EMI for the SSE Lab dashboard. We performed these activities iteratively and evaluated the design and architecture several times with experts and SSE Lab technicians.

During the design phase we noticed missing information (specification) for some important details of the metric that calculates statistical information on the time that a ticket is open. Most importantly for these metrics, it was not defined at what point this information is evaluated and to which interval the information belongs. As an example, lets assume ticket t_1 is opened in week w_1 and stays open for two weeks. Hence it is closed in week $w_1 + 2$. The opening duration of the ticket is two weeks and because there is no other ticket this is also the average, maximum, and minimum opening duration of tickets in this time frame. When a metric customer requests a chart (for example containing boxplots) over this information for the weeks w_1, $w_1 + 1$, and $w_1 + 2$ at which point does the system show what data? Throughout our discussion we identified three different scenarios[6] for this. All of the scenarios have strengths and weaknesses related to the

[6]See the thesis from Arthur Otto for additional details on this subject and the design of the EMI [Ott13]

specific environments and the specific questions of the metric customers. Therefore, we decided to implement all three variants as separate metrics. Thus, the metric customer can select the method that best fits her specific need (at that time).

Based on these results we iteratively and incrementally constructed the data adapters, the integration layer (messages) and the metric kernels. We started with the integration and data adaption on the version control systems and then moved to the issue tracker (TRAC). We also implemented a special dashboard application which can be integrated seamlessly into the SSE Lab frontends. After each increment we did a quick evaluation of the metrics and dashboard in a test environment.

13.2.2. Architecture

The EMI for SSE Lab is depicted in figure 13.8. Contrasting the layout of the reference architecture the data flow in this figure is turned 90° going from left to right. This layout also reflect the pipes-and-filter like nature of the metric calculation inside an EMI [HW03b].

The three central data providers: git version control system, TRAC issue tracking, and sonar qube for source code metrics are located on the left. The data adapters for these systems implement the push-forward adapter pattern (see section 5.4.1). We build specific plugins for TRAC and sonar which hook into existing extension points in the two systems to call the respective REST-API in the data adapter on data change in the data provider. For git we used a bash script that is called in a commit hook to call the REST-API of the commit gateway.

The SSELab EMI utilizes three busses from the EMDB. The event topic (`EMI.events`) to transport event data, the base topic (`EMI.base`) to transport almost raw data (`base measures`) from the data providers, and the measures topic (`EMI.measures`) to transport measurements (derived measures).

The commit event gateway on the top left emits commit event messages to the event topic of the EMI. These messages are transported to two metric kernels: Commit Reference Kernel and Event Counter. The latter is a generic component that calculates a number of count metrics (number of event Y per X) on events on the event topic.

The Commit Reference Kernel simply checks the commit message for references to issues and does not store any data (just static checking, no semantic check!) and then again utilizes the event counter with a different event to count the commit messages without references. This shows the flexibility and reuse potential of the EMI microservices.

The TRAC ticket data rest gateway produces `ticket messages` on the base topic of the EMI. These tickets are analyzed by the TRAC Kernel. This kernel stores the tickets and implements the count metrics (e.g. open tickets per week) as well as the statistical analysis that feeds the box plots with all three variants as discussed above. The metric calculation results as well as the calculations from sonar are feed to the measures topic of the EMI and stored in the measurement cache.

As mentioned before, we built a special metric-based monitoring dashboard to visualize the metrics in SSELab. The dashboard accesses the metrics from the metric kernels using their indicator access APIs and feeds the calculation results to the visualizations.

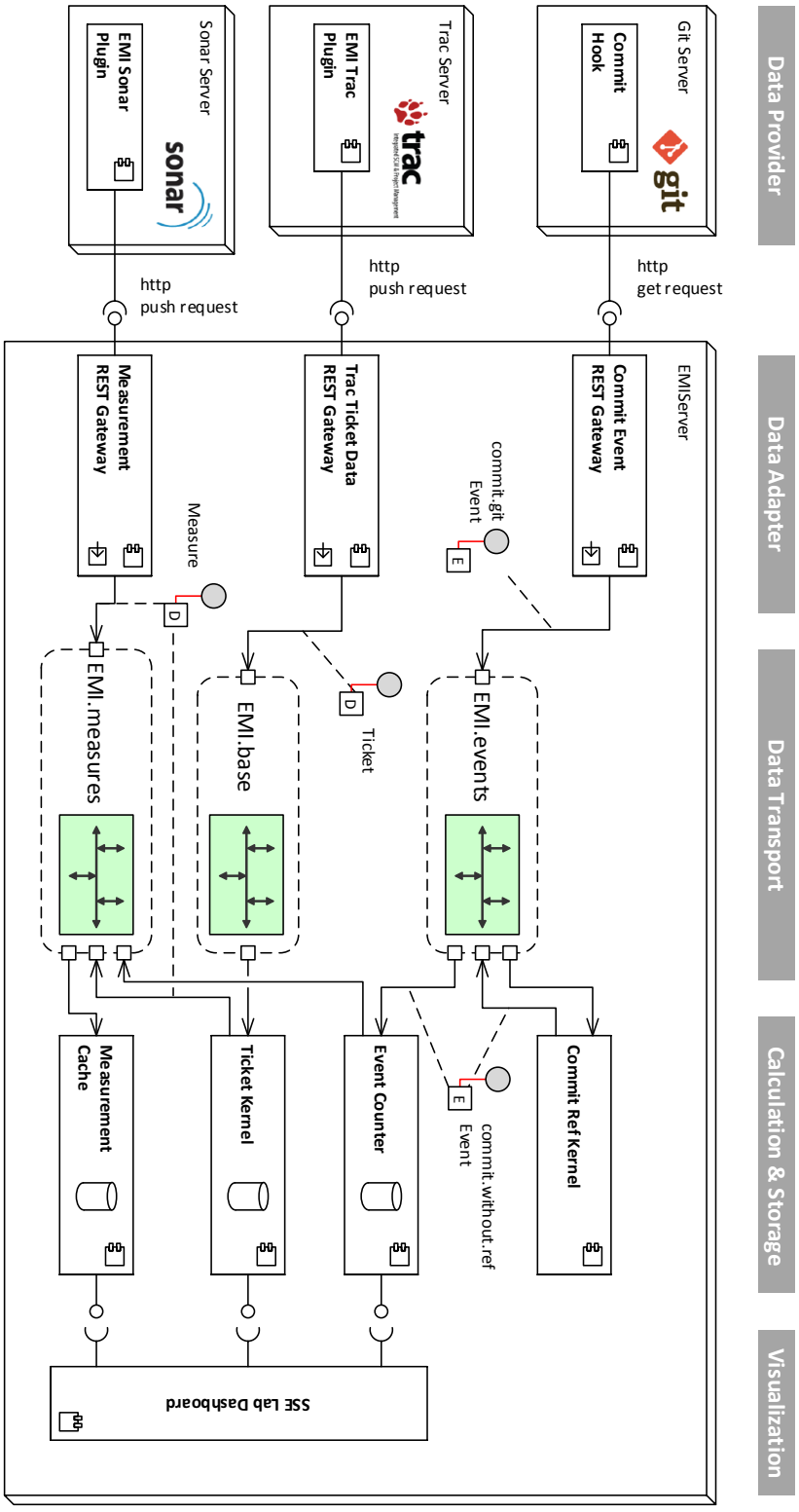


Figure 13.8.: Static architecture overview of the EMI core for the software project metrics in sse lab.

The pre-production version of this EMI was operated on a JavaEE server in our test environment which also hosted the local databases for the metric kernels. This server was operated in a secure environment to ensure data privacy. Thanks to the EMI Monitoring Service it was always very easy to check the current status of the EMI.

13.2.3. Experience

Figure 13.9.: Screenshot of the realization of the SSE Lab metric-based monitoring dashboard

This field study provided a lot of experience on using our engineering approach on the engineering of a metric system for an academic and software development environment with small teams. The core expiries were:

Finding the right metrics is hard! Even though we did extensive prototyping and conducted several interviews with the metric customers (SSELab project managers) we found some problems with the metric specifications when we started designing the metric kernels. Particularly the statistical ticket analysis in the Ticket Metric Kernel was not well defined and we were struggling with what alternative to use. In the end the flexibility of the reference architecture allowed us to implemented all the different options in the metric kernel side-by-side and allow the user to select the calculation mechanism. However, this shows the importance of all the evaluation gates in our process model.

Microservices in an EMI are very easy to reuse! We developed the Event Counter and the Commit Event Gateway before we started the development on the SSELab EMI. During the specification of the EMI for SSELab we realized that we can reuse these services and it worked fluently without any major issue! It was also very easy to add additional functionality (counting commits without references) by simply adding another service.

An EMI is easy to maintain and to operate! During the iterative and incremental development of the EMI for SSELab it was very easy to extend and maintain the services. Thanks to the EMI monitoring service it was always very easy to check the status of each service and to investigate performance of the services and the EMI.

No major performance problems! We never experienced any problem with performance whatsoever because we also never experienced a lot of messages on any of the EMI-topics.

13.3. Flow-based Visual Ticket Analysis

Research Question (Q5.1):

> Are the MeDIC process model and MeDIC reference architecture suitable to engineer analysis systems?

Focus:	Reference architecture
Environment:	Two large IT service providers
Code size: Project specific	< 20k LOC (v1.3) and < 10k LOC (v2.0)
Code size: Overall incl. EMI Framework	< 40k LOC (v1.3) and < 30k LOC (v2.0)
Overall Effort:	≈ 15 PD (+ one bachelor and two master/diploma theses)

During our work with two large IT service providers for insurance companies we noticed that a lot of the information needs of the process managers can be answered with metrics and visualizations based on ticket data. The process started as usual and we gathered some information needs from the process managers. Just providing the different metrics, however, is too much information and too complex; the analysis needs to be guided. Hence, we had the idea to build an analysis tool that shows metric-based monitors for the ticket flow through the edges of the process next to the graphical representation of the process similar to the control center of a chemical plant. Sadly, the first prototypes already showed that this becomes very complex very fast. We only experimented with small ticket status graphs and even with these it was hard to get monitors next to the edges. Hence, we realized we require another graphical representation that shows the amount of flow through the edges more intuitively.

We then decided to use Senkey diagrams. A Senkey diagram is a representation of a weighted acyclic directed graph. A node in the Senkey diagram represents a ticket state, for example open or closed, the size of the node is proportional to the amount of tickets in the state at that position. The size of the arrows between the nodes is proportional to the number of tickets that flow from one state to the other. Because the Senkey diagram visualizes the flow of the tickets we called the analysis tool *RiVER*[7].

13.3.1. Key Concepts

Figure 13.10 shows the Senkey diagram for a data set from one of our cooperation partners. Following the typical process in a ticket management system, the majority of the tickets start in the *New*-state. Approximately $\frac{1}{3}$ of all the tickets then do not have

[7]The first version of the RiVER analysis tool was build in the diploma thesis of Christian Charles [Cha12]. After that Endri Gjino extended the tool with additional visualizations and a data mining component [Gji13]. Finally, Sebastian Rabenhorst significantly improved the performance with the second version of the tool [Rab15].

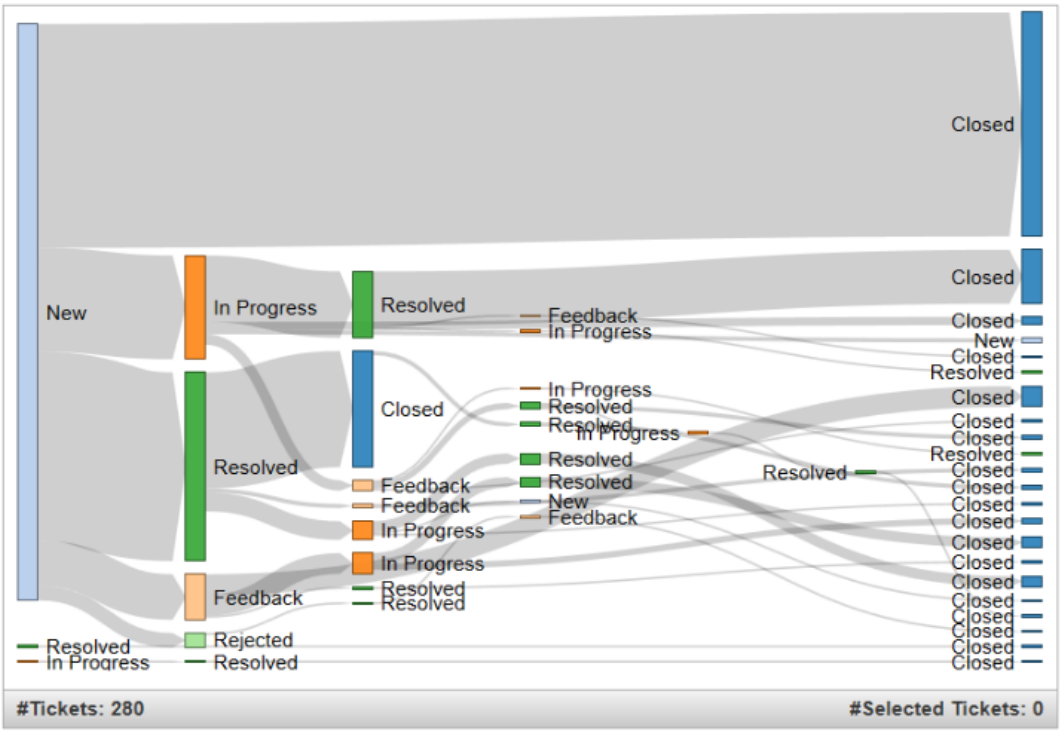


Figure 13.10.: Ticket flow visualized as Senkey diagram in the RiVER analysis tool (taken from [Cha12] p.75).

any intermediate states; their next state is the *Clossed*-state (top arrow). This already indicates a problem because these tickets provide very little details to the actual work performed and do not follow the defined process. Half of the remaining tickets directly flow from the *New*-state to the *Resolved*-state. In the underlying process the tickets should enter the *Resolved*-state when the development is finished and the bug, fix, or feature needs to be evaluated by the customer. The customer may then close the ticket if the evaluation is positive, which half of the tickets actually do. However, $\frac{1}{4}$ flows back into the *In Process*-state or the *Feedback*-state, which indicates that there was a problem with the corresponding work item. Additionally, the remaining $\frac{1}{4}$ of the tickets remain in the *Resolved*-state and are not closed.

This short analysis of the diagram in figure 13.10 already shows the strengths of this diagram: It provides a lot information on a very small space. However, it requires training to read the diagram and it obviously only provides indicators for problems which require a more thorough analysis to find the cause.

After our first evaluation with the metric customers (the process managers) we noticed that the RiVER tool lacked an overview over core problems with the ticket process. The Senkey diagrams were able to show some of them. However, the metric customers needed to select the appropriate filters and data to see specific problems. Therefore, we added

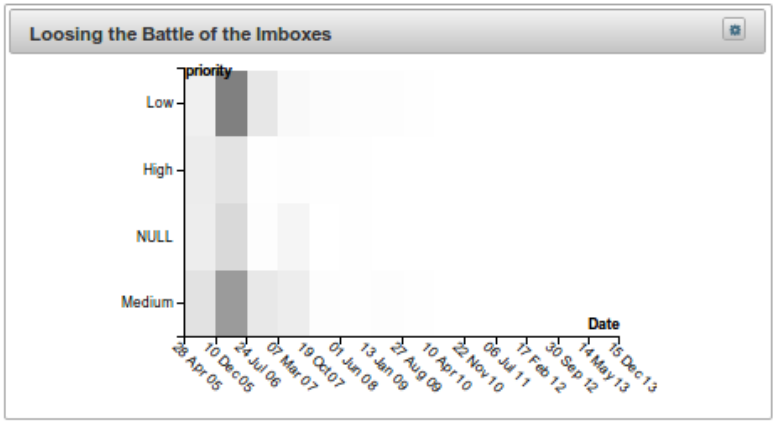


Figure 13.11.: Indication of the smell "loosing the battle of the inbox" using a radiogram visualization for the overview page of the RiVER analysis tool (taken from [Gji13] p.76).

a data mining component to the tool that is able to use classifier to show an overview over core problems in the ticket process. We identified four major problems in ticket management systems [Gji13]:

Zombies Zombie tickets are tickets which remain in an open state and are never closed (like the resolved tickets from above).

Hot Potatoes Hot Potato tickets are tossed around between different components or assignees because no one takes responsibility.

Close-Reopen Cycles Reopening tickets is not necessarily a bad thing but it should not happen more than once.

Loosing the battle of the inbox The assignees are loosing the battle of their inbox when more new tickets are opened than existing tickets are closed.

The classifiers assign a value between 0% and 100% for each smell to each ticket. These values are then aggregated, for example based on the priorities, and visualized using Radiograms, which provide a good overview over potential problems and their development. Figure 13.11 shows one of these Radiograms, which indicates a "loosing the battle of the inboxes"-problem for low priority tickets between December 2005 and July 2006. The metric customers are then able to select a particular region of one of the radiograms and use this region as a filter for the tickets visualized in the Senkey diagram.

The evaluation also showed that the metric customers like to dig a little bit deeper into the actual ticket data when selecting an edge or node in the Senkey diagram. Therefore, we added a work-item-history visualization for all the tickets in the current selection. Figure 13.12 shows an example for this. Contrasting the Senkey diagram, the work-item-history uses time as the x-axis, which allows to evaluate the time between the status changes. Furthermore, we reduced the ticket states to just *open*, *resolved*,

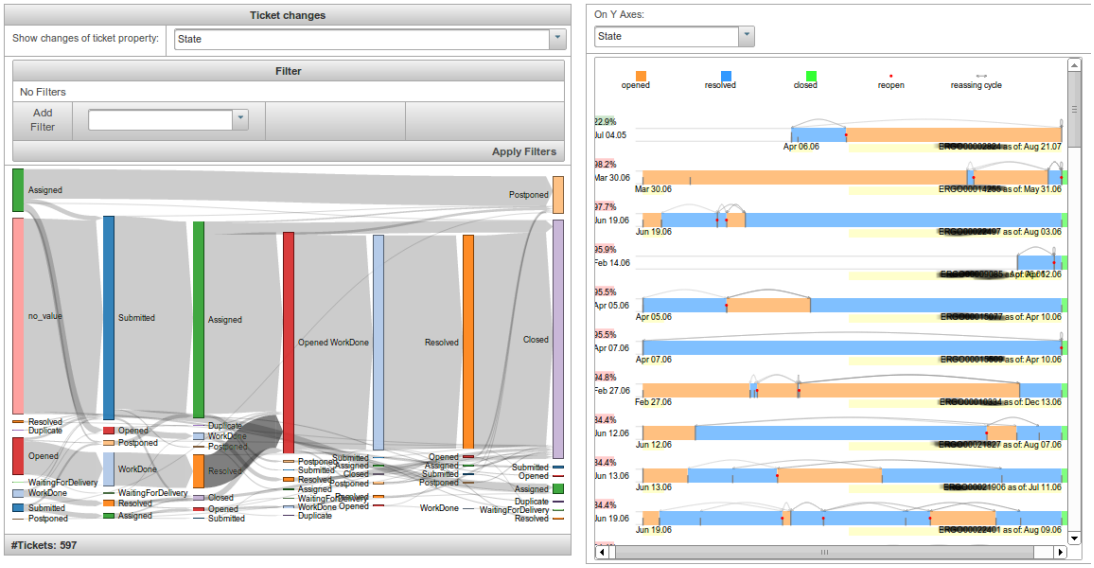


Figure 13.12.: Specific ticket status changes visualized as work-item-history in the detail view of the RiVER analysis tool (taken from [Gji13] p.77).

and *closed*. This helps the process managers to focus on the important parts of the underlaying ticket process and avoids visual clutter with too many colors.

13.3.2. Architecture - First Version

The engineering of the RiVER analysis tool (and the underlying metric system) followed our metric systems engineering process model. Therefore, the architecture of the tool is also based on our reference architecture. Throughout this field study, we built two versions of the tool. The first version was built in two development increments. After each we evaluated the tool with multiple process managers from multiple companies. Yet, we noticed major performance issues and usability problems with the tool after the second evaluation. Therefore, we rebuilt the tool in a second version. This section discusses the first version.

Figure 13.13 depicts the EMI for the first version of the RiVER analysis tool. This version used three different invoke-dump type data adapters (see section 5.4.1) to adapt Redmine and Trac ticketing system as well as an universal CSV adapter that we mainly used to adapt data from Rational Clear Quest. The adaption is triggered by the `RiVER Config` component in the visualization layer. This component allows the configuration of the data sources and the upload of the cvs files. From this a data adaption is triggered via messages on the `EMI.command` topic. The `Dump Commands` contains the configuration for the `Redmine Adapter` and the `Trac Ticket Adapter`. The `Adapt CSV Command` contains the configuration as well as the actual CSV-file for the adaption via the `CSV Ticket Data Adapter`. All of these data adapter send `Ticket Base Data` and `Ticket Status` messages via the `EMI.base` topic.

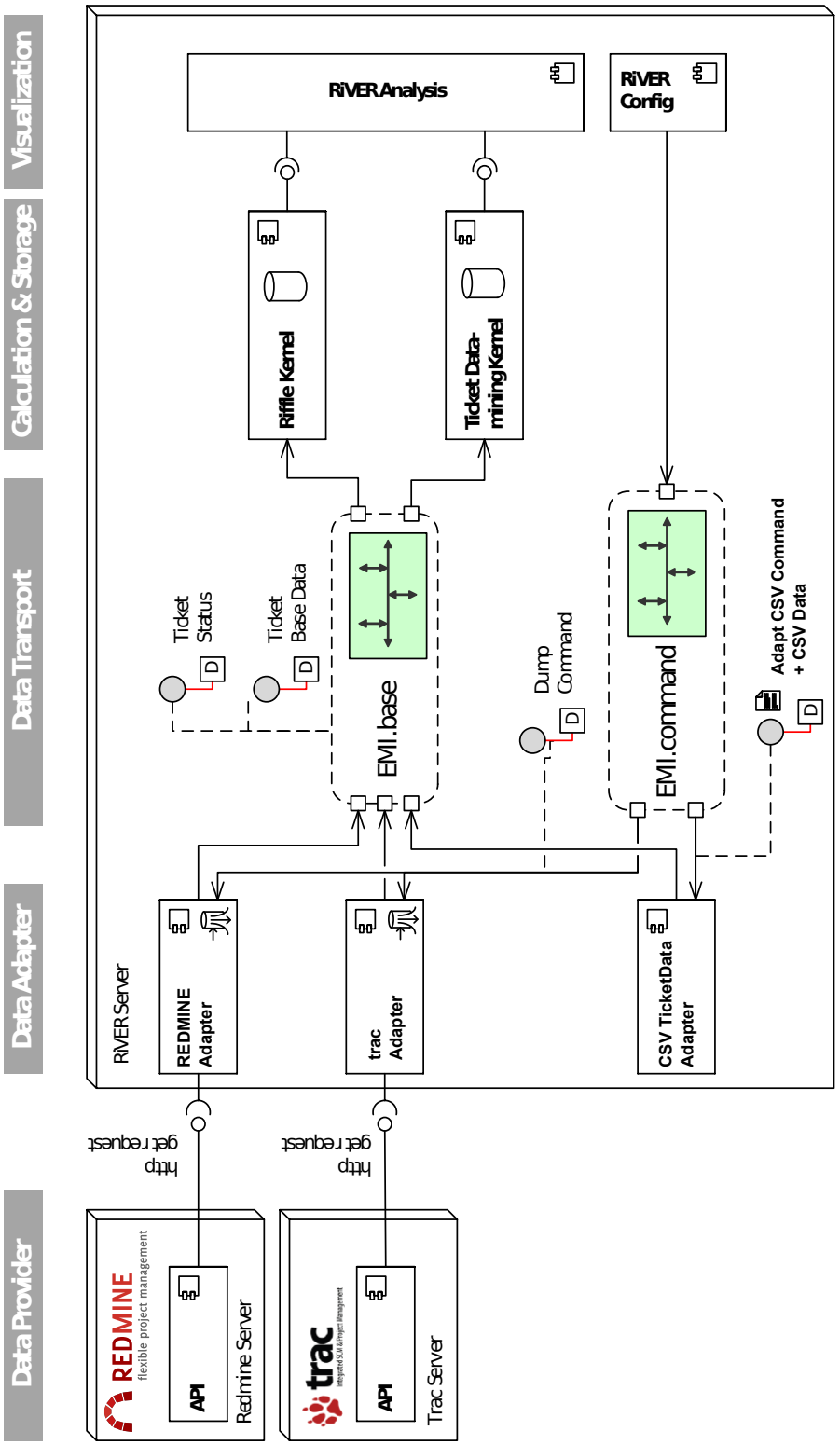


Figure 13.13.: Static architecture overview of the first version of the RiVER analysis tool.

The base data messages contain all (initial) values of a ticket whereas the status messages only contain the changed values. This reduces the size of the messages because not each status message needs to contain all fields. The two metric kernels realized for this version are: first the `Riffle Kernel` which provides the indicators for the Senkey diagrams and ticket details, second the `Ticket Datamining Kernel` which provides the Radiograms and two prognostic indicators. The indicators from these two kernels are then consumed by the RiVER analysis tool, which provides the fixed visualization frontend of the tool.

Both metric kernels use a relational database as persistent storage for their data. Both kernels store the raw ticket data and status history for each. Hence, the calculations (and transformations) to provide the indicators for the Radiograms and the Senkey diagram are performed upon request by the visualization frontend. The rational behind this design decision was the implementation of a very flexible filter mechanism for the data of the indicators used to focus the data in the diagrams.

The tool performed well in our initial evaluations with small and medium sized data sets (up to 10.000 status changes) from different companies. However, we then started to analyze larger data sets from two other cooperation partners that contained up to 1.5 million status changes. We started with smaller chunks of up to 100,000 status changes which the tool was able to consume and analyze. Yet, the visualization was only able to show a filtered subset of these large data sets due to long calculations and transformations of the data in the metric kernels. Therefore, we started the development of a second version of the tool to overcome these issues.

13.3.3. Architecture - Second Version

The engineering of the second version started with a thorough analysis on the performance problems of the first version. We identified the key problem to be the database and persistence technology used as well as the calculation and transformation algorithms which provided the data for the Senkey diagrams. The algorithms provided a graph data structure that was feed to the visualization in the visualization frontend. Therefore, we decided to use a graph database which could easily store multiple and aggregated graphs which could be directly feed to the visualization frontend. With this we then decided to rework the whole analysis tool to also address other usability issues with the first version (navigation, filtering, and additional data analysis tools).

Figure 13.14 shows the Senkey diagram component from the second version. This picture is based on approximately 1.5 million ticket status changes from the Rational Clear Quest system of one of our cooperation partners. We decided to remove the name of the nodes from the diagram to allow a more compact representation. The names are now provided via a legend on the right hand side of the diagram. The legend highlights all particular nodes of the specific status on mouse hovering in order to allow a focused analysis on a specific status. The data filters are now provided as fixed filters for a date range, a specific status that should be included, and a filter on the ticket priority. These filters are located directly above the diagram (not shown in the figure).

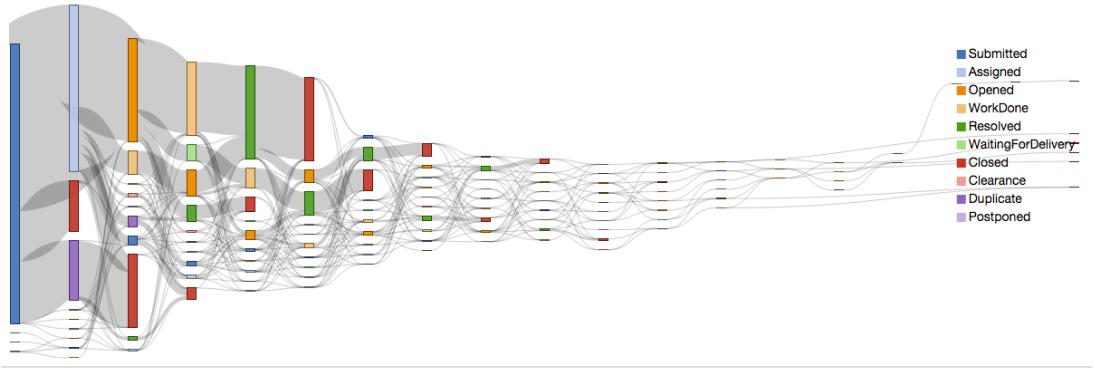


Figure 13.14.: Senkey diagram from the second version of the RiVER analysis tool showing approximately 1.5 million ticket status changes (taken from [Rab15] p.26). We drastically simplified the visualization (e.g. removed the names of the nodes) to make it more compact.

Right next to the main Senkey diagram we placed another Senkey diagram that shows the in- and outgoing flows for the currently selected status node (also not shown in the figure). This acts as a zoom into the main Senkey diagram and eases the analysis of a specific status node. Furthermore, the visualization frontend also provides a list of all the tickets in the currently selected node or flow to enable a more detailed analysis on potential problems similar to the first version of the tool.

The architecture for the second version is provided in figure 13.15. Contrasting the first version, this one only provides a data adapter for the adaption of CSV files because the large dumps from the Rational Clear Quest systems from the two cooperation partners were provided as CSV files. We again triggered the adaption via the `RiVER Config` tool. However, contrasting the first version we did not include the CSV data in the command because it was too large. Thus, the `CSV Ticket Data Adapter` accessed the CSV file from the local file system on the river server. Similar to the first version, this then sends `Ticket Base Data` and `Ticket Status` messages on the `EMI.base` topic. Contrasting the first version, however, the `Ticket Status` messages first entered the `Ticket Normalization` metric kernel which unifies the different states from the different ticketing systems using the directory service. The kernel then sends `Normalized Ticket Status` messages back to the base topic. These messages are then received by the new `Graph-based Riffle Kernel` which sorts the status data into the graphs and prepares the basis for the Senkey graphs. The `RiVER Analysis` visualization frontend, as usual, then accesses the graphs via the indicator access API of the kernel.

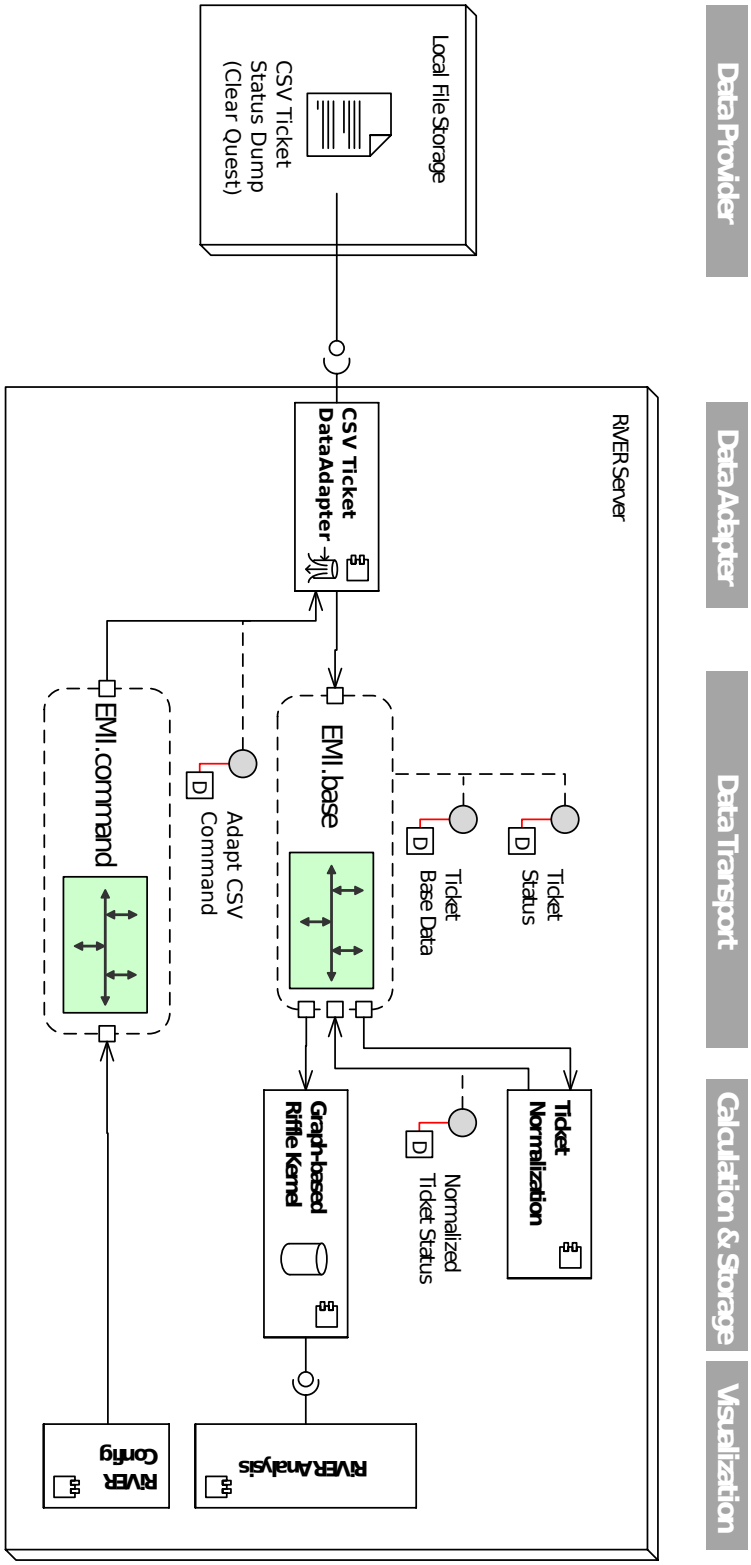


Figure 13.15.: Static architecture overview of the second version of the RiVER analysis tool.

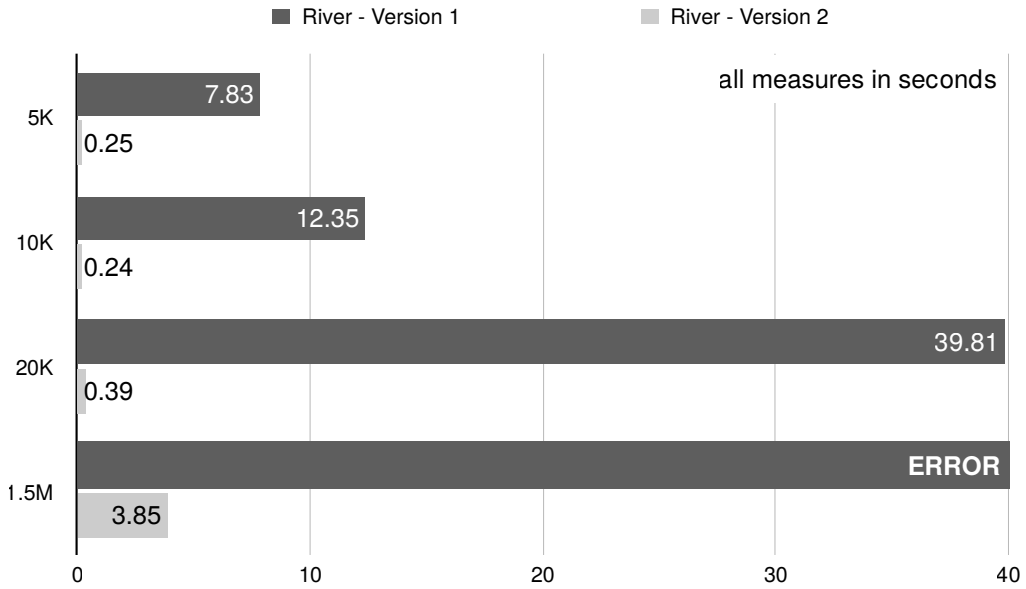


Figure 13.16.: Comparison of the Senkey diagram provision speed of the two versions of the RiVER analysis tool based on the number of status changes in the databases (smaller is better).

Figure 13.16 provides a comparison of the speed to provide the indicators for the Senkey diagrams in the two versions of the RiVER analysis tool. The left hand side shows the number of ticket status in the respective data storage of the Riffle kernels. The speed to provide the indicators is measured in seconds and provided as bar charts next to the status number; the first version is shown as dark gray, the second version is shown as light gray.

The first version is able to handle 5,000 and 10,000 ticket states quiet well with calculation speeds of 7.83 seconds and 12.35 seconds respectively. However, the calculation for the indicator for the Senkey diagram based on 50.000 ticket states lasts 39.81 seconds which already is quiet uncomfortable to work with. The calculation of the indicator for a Senkey diagram based on 1.5 million states is not possible due to the exponential development of the calculation time.

The second version, on the other hand, provides the indicators for the first three state sets (5.000, 10.000, 20.000) almost instantly with 0.25, 0.24 and 0.39 seconds calculation time. The second version also provides the indicators for the large data set of approximately 1.5 million states very fast with 3.85 seconds calculation time. This is faster then the calculation of the indicator for the 5.000 states in the first version.

The increase in calculation speed is manly caused by the optimization of the database technology and the according optimization of the storage mechanism. However, this mechanism pre-calculates some of the graph data on the reception of the status messages in the Riffle kernel which could potentially slow down the reception significantly. Yet,

our analysis shows that the reception and storage speed is almost identical in the two versions with only a little overhead in the second version[8].

13.3.4. Experience

Again, we gained a lot of experience on using the reference architecture for the two versions of the RiVER analysis tool. Furthermore, we gained experience on using MeDIC for the engineering of an analysis tool rather than a metric-based monitoring dashboard. The core experiences were:

Technological flexibility is a key success factor! The performance improvements between the first and the second version of the analysis tool are mainly due to the change in the persistence and database technologies used in the Riffle metric kernel. This again shows the important of flexible selection of technologies for the different components of the architecture which is one of the key aspects of our reference architecture.

[8] See the bachelor thesis from Sebastian Rabenhorst for additional details [Rab15].

Engineering an analysis tool requires multiple iterations! Contrasting the engineering of a metric-based monitoring dashboard, when engineering an analysis tool, the metric customers need to use the actual tool on their data in order to provide feedback. Yet, prototyping is still important. The first prototypes provided a good idea of what not to do (monitors next to the process model). However, the actual evaluation of the analysis tool requires that the metric customers are able to use the real tool with their real data. We believe the reason for this are vague information needs of the metric customer regarding the actual context[9]. Hence, from our experience the engineering of an analysis tool requires multiple iterations, which include try-and-error to a certain degree.

Addressing and improving performance and usability issues is important! Challenge C3 already underlined the importance of usefulness and usability of the metric system. We noticed that the usability improvements in the second version were very much appreciated by the metric customers. Even though the second version provided less functionality than the first version. Most importantly, the performance improvements were a direct necessity for the analysis of big data sets.

Non-standard visualizations are very suitable for an analysis tool! We saw that the Senkey diagrams, as soon as they are understood, are far superior over conventional charts at indicating problems and for the communication of certain issues. For example we visualized the changes (flow) for a bug reason using the diagrams. The same information can very easily be visualized as a bar chart. However, we noticed that people grasped the problems more easily from the Senkey diagram because the flow visualized the data change and not a fixed snapshot. This aligns well with popular best practices to always use the most fitting visualization for the *story* that needs to be communicated [Few12].

However, non-standard visualization require additional training for the metric customers. We provided a workshop to explain the Senkey Diagrams, Radiograms, and Work-Item-History. After this the metric customers could use the tool and the visualizations without problems.

[9] They have the feeling that some things in the process can be improved but they do not know for sure what the problems are.

14 Tooling

Some of the most crucial activities in our process model need to be supported by appropriate tools. Additionally, our field studies required that we built a lot of the services and tools described in our reference architecture. This chapter will briefly present and discuss the tools that we build and used in our field studies.

The chapter starts with the description of two process support tools. Section 14.1 presents a model driven tooling approach for metric documentation. Section 14.2 presents a web-based tool which enables project managers to define and document the metrics used in their project following the requirements of CMMI level 3. In our field studies we used the two tools to document some of the results of the development process and to gather input from the project managers. We then discuss the two different dashboard applications, MeDIC Dashboard and SCREEN, in section 14.3. The last two sections focus on development support for actual EMIs. Section 14.4 will provide an overview over the different EMI support services that we built to implement the different services required in our reference architecture. Finally, section 14.5 presents a brief overview over the EMI-Framework which we used to build the different services for all EMI-related field studies.

14.1. MeDIC Metric Documentation Tools

One of our cooperation partners started an initiative to reach CMMI level 3. During initial appraisals they noticed problems related to metric documentation and metric utilization in the projects. Furthermore, they realized that defining company wide metrics that should be used by the projects is a challenging task. Initially, the company wide metrics have been documented in a single large document. This document was hard to read, hard to navigate, and the single metric descriptions differed in granularity and amount of information. Therefore, we started to develop a model-based approach for metric documentation, which generates an easy to navigate html-based documentation. The tool used a very rich and expressive meta model as the basis for the approach that was inspired by similar approaches in the literature [MGRP09, OLP02, OMF+03, MJCH08].

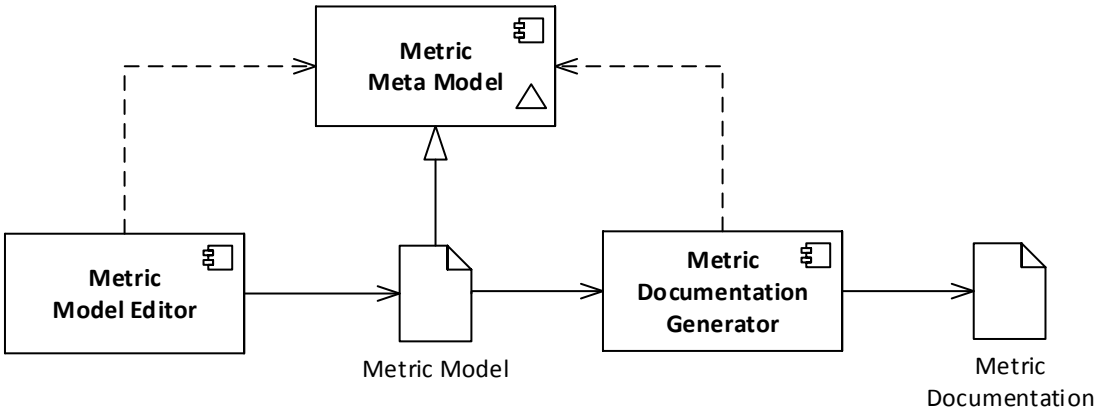


Figure 14.1.: Overview of the modeling workflow of the tool.

Figure 14.1 provides an overview over the modeling workflow implemented by the modeling tool. The metric meta model is an extension of the terminology model for our metric portfolio in section 2.1. The two core components are the metric model editor and the metric documentation generator. The model editor allows the definition of metric models which are instances of the meta model. These metric models are then feed to the metric documentation generator which utilizes a number of model-to-text transformations in order to generate the html-based metric documentation.

Figure 14.2 provides a screenshot of the model visualization in the documentation tool showing an excerpt of an actual metric documentation model. The metrics are defined for their entity of measurement class. In the screenshot this is the class “Großes SE-Projekt” (German for “large software engineering project”). The metrics features three earned value metrics for project management (EV, AC and CPI) [Anb04]. Most importantly, the metric expert is able to document assumptions for the metrics (yellow warning symbols), the model behind the metric, and indicator assessments for automatic warnings if the values of a metric are outside a reasonable range. The tool was built using Eclipse EMF technologies. The generators for the html-based metric documentation used Eclipse Xtent and Xpand technologies for the model-to-model and model-to-text transformations.

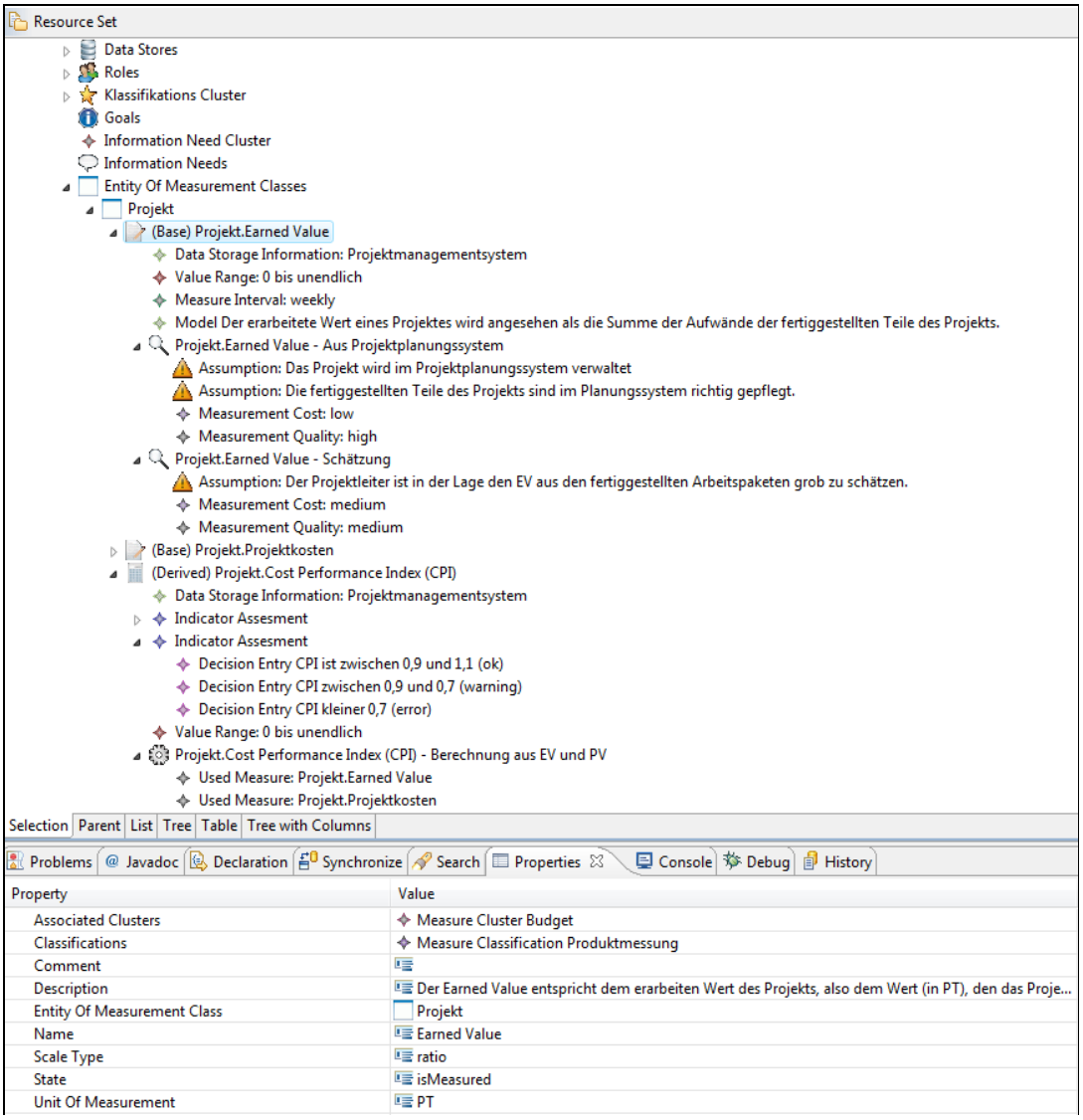


Figure 14.2.: Screenshot of the metric documentation model in the documentation tool.

Figure 14.3 provides a screenshot of a catalog web page of the generated metric documentation. Most importantly, metric customers can browse this documentation based on their actual information needs (questions). These questions are organized in different categories and associated with different roles of metric customers. Hence, metric customers are able to quickly find suitable monitors to answer their needs. The give screenshot, for example, provides a monitor which visualizes the execution and definition of test cases in order to address the information need “How is the product quality developing based on test cases?”.

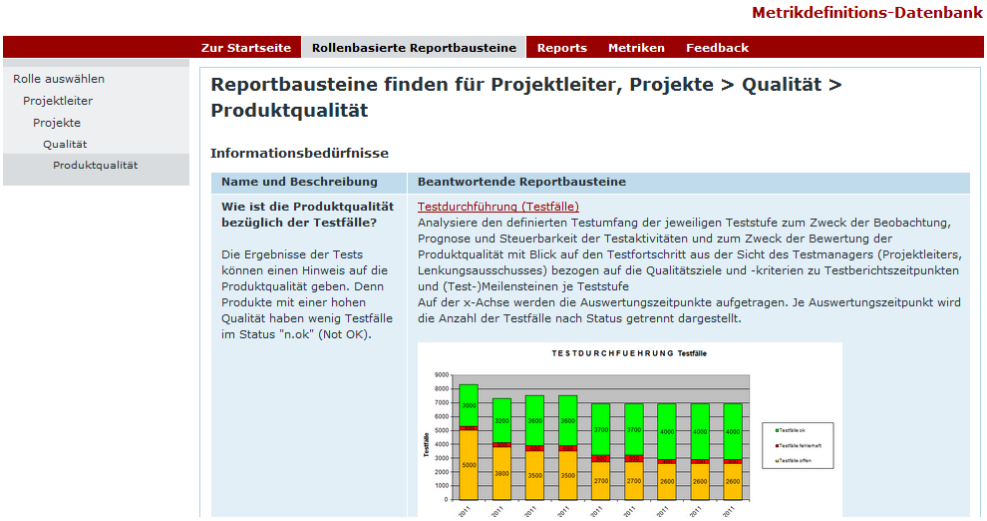


Figure 14.3.: Screenshot of a web page of the generated metric documentation.

One major problem with the tool in an industrial environment, however, was the complexity of the underlying metric model. Based on the related work and our other experiences we added numerous attributes and entities to the model which need to be specified. Therefore, modeling all these different properties was a laborious task and very tedious. Our interviews with project managers and other metric customers also indicated that all these additional informations provided no additional benefits for them. Thus, all our other tools drastically reduced the complexity of the metric model to the entities in our metric portfolio (see section 2.1).

The metric customers very much liked the idea of the information need driven monitor catalog which helped them to quickly find suitable metrics. We also linked the monitor description pages from the initial dashboards. Therefore, the metric customers were able to quickly look at additional information regarding a specific monitor if they are unsure about the interpretation or require additional information about the visualized metrics. The metric experts very much liked the ability to add assumptions to the metrics in order to provide additional information about appropriate usage of the metrics.

Most importantly, however, the tool lacked the ability to support the interaction between the metric customers and the metric experts because it provided only static documentation pages. We addressed this issue with a dynamic web-based information system which is described in the following section.

14.2. MeDIC Metric Management Support Tool

While working with our cooperation partners we notices the lack of suitable support for project specific metric definition in a CMMI level 3 environment. CMMI level 3 requires the project managers to define the metrics from the company wide standard they use. Additionally, they need to fully specify project specific metrics. For each of these they need to provide a number of properties in order to fulfill the requirements of CMMI level 3. Therefore, we extended the ideas of our meta model based metric documentation presented in the previous section in order to provide suitable tooling for this.

Figure 14.4.: Screenshot of the overview page for a project in the metric management support tool.

Figure 14.4 shows a screenshot of the overview page for a project in our metric management support tool. This page shows all the monitors which the project is currently using as well as their status. In this particular case, the project utilizes four different monitors (“Projektgeschwindigkeit”, “Scopeindex”, “Number of use case steps”, and “Testdurchführung”) each in their own category. The definition for all of these monitors is not complete as indicated by the warning icon in the “Def” column.

However, this is intended, because all monitors are still being defined and not used as indicated by the document icon in the “status” column. Additionally, the source of the monitors is shown in the “Type” column. The first metric “Projektgeschwindigkeit” is a project specific metric (indicated by the “P” icon) whereas the other three are instances of standard monitors (indicated by the “S” icon). Users with sufficient rights can use the link on the bottom of the page (“Weitere Metriken verwenden”). This will bring up a tree of company wide defined monitors with the status “Best-Practice” or “Mandatory” in respective categories. The user can then select additional monitors for this project and use them. Most of the company wide defined monitors, however, need to be further specified. For example the reporting and measurement interval needs to be defined because this varies drastically between different projects and project phases. Additionally, complete new project specific monitors can be defined.

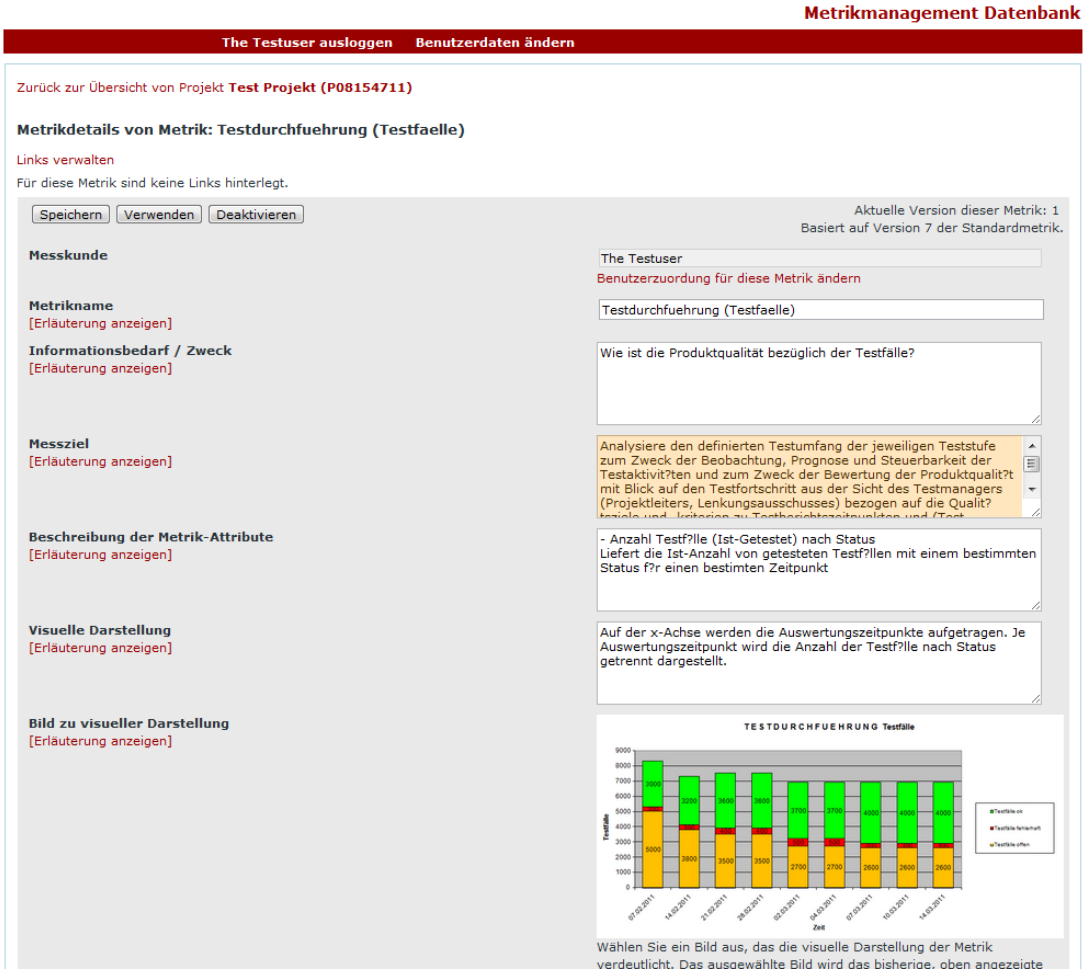


Figure 14.5.: Screenshot of the edit page for a monitor and its metrics in the metric management support tool.

Figure 14.5 shows the editing page for a monitor and its metrics. Each monitor is described via a set of 18 properties; the screenshot shows the first 7. The properties are directly taken from the CMMI level 3 requirements for the process area "Measurement and Analysis". Most of these properties are text; except the description of the visualizations which requires a picture from a mockup or an actual screenshot of the monitor.

The monitor in the screenshot is an instantiation for a company wide standard monitor. Hence, changes and adoptions need to be indicated. The fourth property "Messziel" (goal of the measurement) is highlighted with a different color because the value of this property is changed with regards to the company wide standard. In order to justify the change the user can add a justification description in an additional text field which can be opened on demand. On the top right hand corner of the gray area for the properties versioning information for this monitor is shown[1].

The tool supports the metric expert with a lot of valuable inside into the metric and monitor usage of the different projects. They are also easily able to extend the company wide catalog of standard monitors and to enhance existing monitor definitions. Furthermore, the tool supports the monitor seniority process in which monitors go through different status. They start as an "idea" in a project with an initial description. When the monitor is utilized and developed it is marked as "individual" or project specific monitor. From there it can enter the company wide standard as optional "best-practice". When it turns out that the monitor is working exceptionally well it can also be marked as "mandatory". When the monitor is no longer needed it can be marked as "deprecated" and projects who use the monitor get a warning for that particular monitor. If no project uses the monitor anymore it can then be "archived" in order to keep a record of the monitor. From there it can be revived into "best-practice" state again if it turns out that the monitor is again useful.

The tool was developed as a java enterprise edition information system in close coordination with one of our cooperation partners. The first version of the tool was developed as part of the bachelor thesis of Frederic Evers [Eve10]. We then further enhanced it in a scrum like process with customer involvement every three to four weeks. The tool then entered a pilot phase at the customers in which it was further evaluated at the company. However, it was not used in production after this because of missing budget, personal, and responsibilities for the operation of the tool, even though our evaluation should overwhelming satisfaction and need for the tool at the stakeholders.

[1] In this case, the version of the project specific instance of the monitor is version 1. The monitor is based on the version number 7 from the company wide standard monitor.

14.3. MeDIC Dashboard and SCREEN

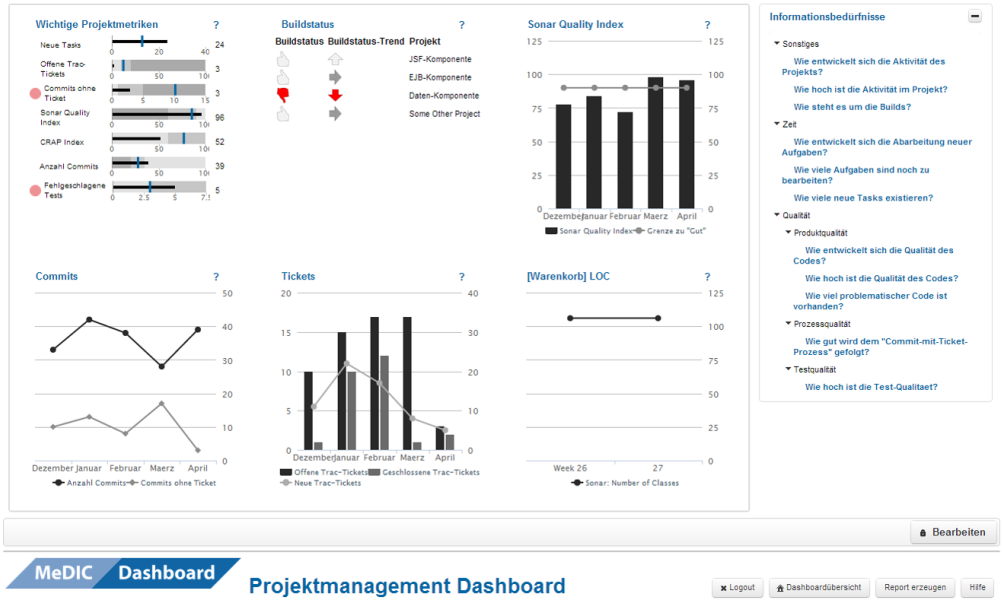


Figure 14.6.: Screenshot of the dashboard frontend from MeDIC Dashboard.

The first two tools focused on metric and monitor documentation and general support for the metric management process. With these and our growing industry cooperation projects we did realize the need to support the measurement infrastructures as discussed in this thesis. Our work on this started with the MeDIC Dashboard (see screenshot in figure 14.6) and continued with our second dashboard application SCREEN. The goal of those two applications was to provide flexible dashboard tooling which embodies the ideas for modern dashboard design [Few06]. The dashboards are fixed to one screen with automatic resizing and stretching of the dashboard and its monitors. Also the color pallet of the widgets is drastically reduced in order to help the user to focus on important warnings which are indicated using red.

The dashboard could be edited using drag&drop for the monitors. In MeDIC Dashboard the monitor editing and placement, however, required some detailed editing in the backend of the dashboard which was more focused towards metric experts. SCREEN used more user friendly dialogs and advanced templating mechanisms. These were partially placed in other services, following the micro-service ideas, to allow SCREEN to focus only on the dashboard management and interaction. Most importantly, the monitor selection for a particular dashboard spot in SCREEN used a tree of questions similar to our metric documentation tooling from above. This is particular useful for unexperienced users who do not necessarily know what monitor to use to answer specific questions in their context.

The two most important aspects of MeDIC dashboard were the always visible information need sidebar and an advance collaboration mechanism for the interpretation

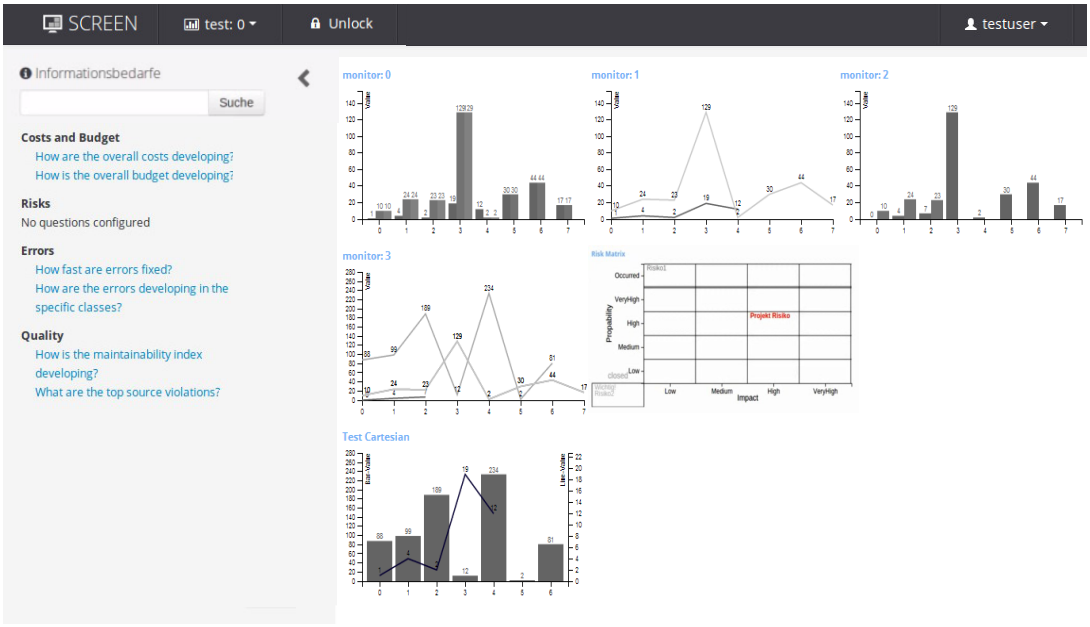


Figure 14.7.: Screenshot of the dashboard frontend from SCREEN.

of monitors. The information need sidebar showed all the information needs which are answered in the dashboard (see screenshot). Hence, the user could quickly glance at this sidebar to get an idea about the questions which are answered in the particular dashboard. Our field studies, however, showed that more advanced users do not really use and need this sidebar. They rather use the space for the sidebar to place additional monitors or enlarge existing once. Consequently, we kept the sidebar in SCREEN but it is hidden as a default. The user can expand the information need sidebar in SCREEN by clicking a blue info banner on the left hand side (see screenshot in figure 14.7). Furthermore, we enhanced the interaction with the questions in the sidebar. When users hovered their mouse over a question SCREEN highlighted the particular monitor(s) which answer the specific question. Both tools allowed to enlarge a particular monitor in an overlay by clicking its headline or the particular question.

As mentioned above, in MeDIC Dashboard we also included an advanced collaboration mechanism to annotate monitors with highlights, arrows, and text and to share these interpretations with other users. Our idea was to use this as a collaborative approach to interpret metric data and to share project knowledge. However, we experienced little use and excitement for this feature with our cooperation partners. Therefore, we dropped it in SCREEN. Additionally, SCREEN offered advanced dashboard management features. Dashboards can belong to groups in order to share them among metric customers for project reporting and communication. These group dashboards can be used as the basis for individual metric customer dashboards. Additionally, Metric Experts are able to provide dashboard templates to ease the setup of dashboards by metric customers.

14.3.1. Architecture - MeDIC Dashboard

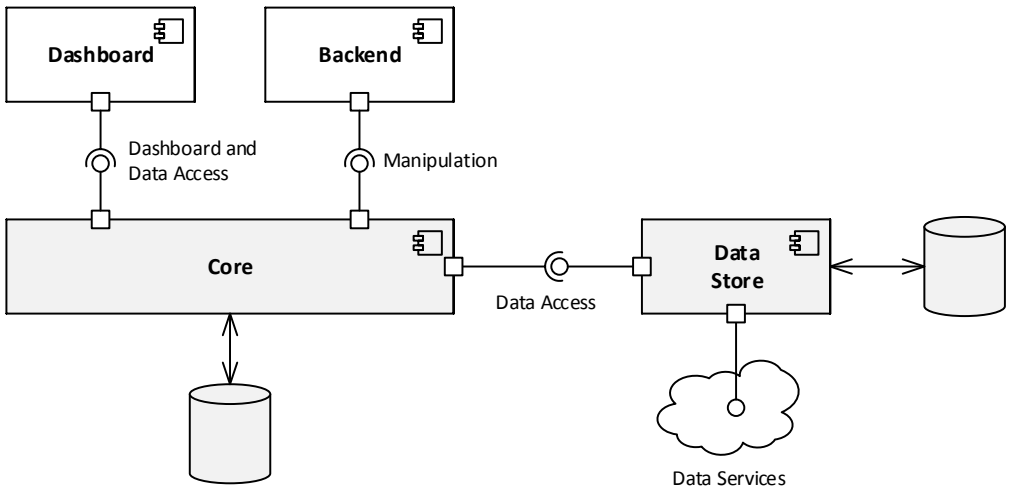


Figure 14.8.: High level architecture overview of MeDIC Dashboard.

Figure 14.8 provides an overview over the architecture of MeDIC Dashboard. The architecture was rather monolithic and does not follow our reference architecture[2]. The dashboard frontend itself is build as a single JSF application which includes all the different renderer and visualizations. This applications connects to a EJB-based core, which provides dashboard management functionality. The manipulation of the dashboards (except the monitor placement) was realized using a dedicated JSF backend-application which also connects to the core.

The core was flanked by a data store component. This provided the data store of the measurement data that was displayed on the dashboard. The measurement data was feed to the data store using two web-services. The first web-service accepted timeseries values (a measurement + a timestamp). The second web-service accepted arbitrary key-value pairs. All the data was then stored in the database connected to the MeDIC Dashboard application. The monitors on the frontend accessed the data from the database via the core.

The design is rather simplistic and resulted in a lot of maintenance and usability issues. However, we were able to use this application in our first field studies and the tool was used in industrial contexts[3]. These field studies provided valuable inside into the usage of the dashboards and further focused our design goals. They also proved our critics concerning the quality problems with the application, which were the reasons why we rebuilt the dashboard application and started to design our reference architecture.

[2]We developed our reference architecture after we build MeDIC dashboard as a result of our experience with it.

[3]For example see bachelor thesis from Christian Hans [Han12]

14.3.2. Architecture - SCREEN

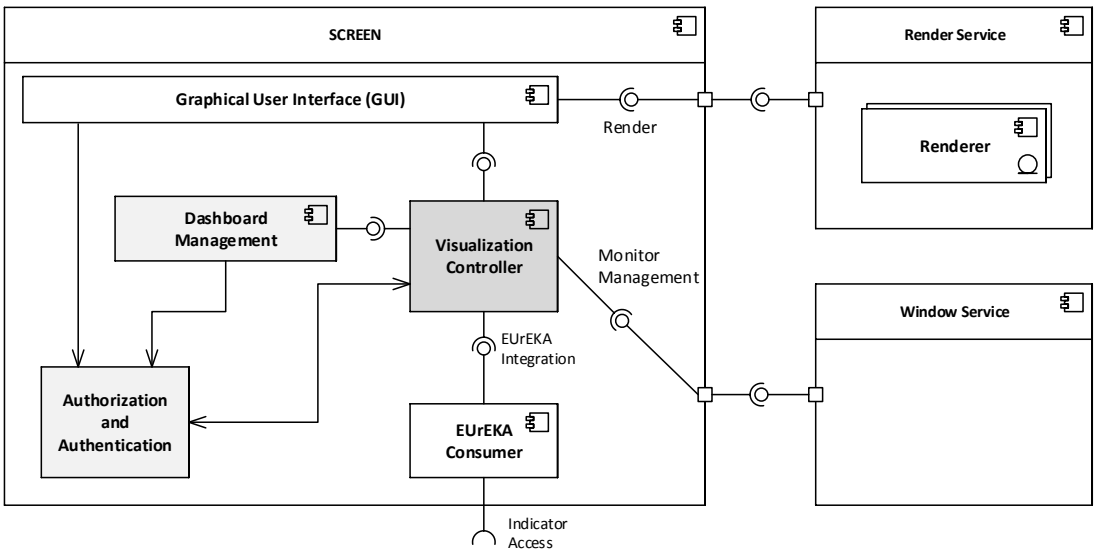


Figure 14.9.: Architecture overview of SCREEN and its connection to the Render-Service and the Window-Service.

Figure 14.9 provides an overview over the architecture of SCREEN and its connections to the Render-Service and the Window-Service. SCREEN follows a micro-service based architecture style that also provides the basis for the EMI reference architecture. SCREEN itself provides the dashboard management functions as described in our visualization reference architecture in section 5.6.3. The measurement data for SCREEN is provided by a EUrEKA Consumer which accesses metric kernels in an EMI as described in section 5.18. It is flanked by the Render-Service on the right hand side which provides the renderer for the different monitors on the dashboard. This enables a very flexible, iterative, and incremental extension and enhancement of the rendering possibilities of SCREEN. Also on the right hand side, SCREEN is flanked by the Window-Service which provides the monitor management and monitor templates. SCREEN uses this service to store the monitor configurations of each dashboard. Furthermore, the service provides the monitor templates and links them to the questions which are featured in the sidebar and the monitor instantiation dialog. Hence, the most benefits for metric customers are provided by the window service and dashboard templates in SCREEN which are maintained by the metric experts.

The SCREEN architecture was designed for extensibility and flexibility. Hence, we were able to easily extend the functionality of SCREEN and add additional features. For example we added a mechanism for the generation of multi-project dashboards to SCREEN which automatically generates multi-project dashboards on-the-fly based on the definition of the multi-projects and some additional monitor configuration. Thus, the dashboards always reflect the current setup of the multi-projects without any additional

configuration. This extension simply hooked into the Dashboard Management component in order to automatically provide the dashboards. Additionally, we only needed to add the links to these dashboards in the GUI component of SCREEN.

All this flexibility allowed us to specifically tailor SCREEN to the needs in the particular field study and easily adapt to new requirements. However, the price to pay is the increased structural complexity, orchestration, and configuration of all the different services. Furthermore, SCREEN utilizes a EuREKA Consumer to access the data for the dashboard. Therefore, SCREEN should be embedded in an EMI in order to use its full potential.

14.4. EMI Services

Our technical reference architecture from part II provides reference architectures for two different types of services in an EMI. The first focuses on services which implement the different measurement functions in the EMI. These are: Data Adapters, Metric Kernels, and Visualization Frontends. The second focuses on services which are only required once per EMI. These services are for example the operation services (monitoring, logging, and directory – see chapter 6), and the Render Service for SCREEN. We will focus the following discussion on these four because they where used most frequently in all the EMIs that be built. Furthermore, they provide interesting discussions on their GUI. Other services, specifically the EUrEKA Services (Registry, Producer Gateway, Indicator Wrapper, and Consumer), are very technical. Hence, their description would, in most parts, echo the description of the services in the reference architecture.

14.4.1. EMS - EMI Monitoring Service

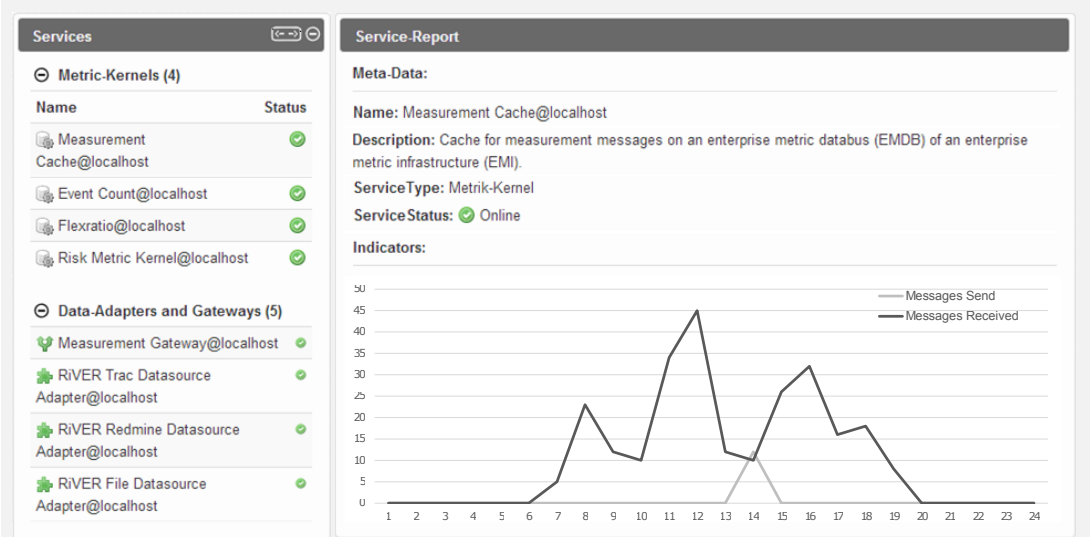


Figure 14.10.: Screenshot of the EMI Monitoring Service graphical user interface.

Figure 14.10 provides a screenshot of the graphical user interface of the EMI Monitoring Service. All the services in the EMI, on which the EMS operates, are shown in the list on the left hand side. The list is sorted by the different types of services (Data Adapter, Metric Kernels, and Others). The service provides a status indicator next to each service in order to quickly monitor the status of each service. On click on a service name, the GUI shows additional details to the particular service on the right hand side. Each service provides a name and a description as well as a status. Furthermore, it lists all the indicators which the particular service provides. In the screenshot the service provides indicators for send and received messages as a combined line chart.

14.4.2. ELS - EMI Logging Service

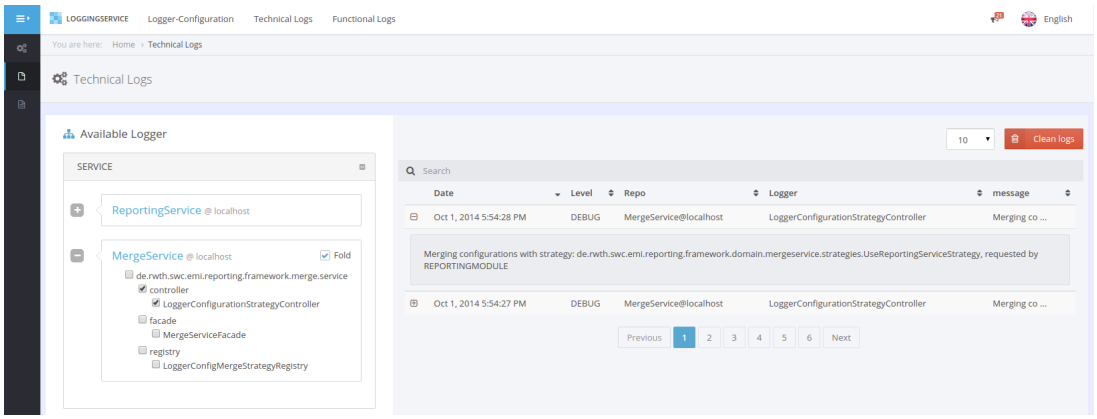


Figure 14.11.: Screenshot of the technical log view in the GUI of the EMI Logging Service (taken from [Dör14] p. 71)

Figure 14.11 provides a screenshot of the view on technical log information in the GUI of the EMI Logging Service. The top and left of the screen shows the navigation on which the three main areas of the ELS can be accessed: technical log view, functional log view, and logger configuration. The area in the center shows a tree-based filter for the services and their logger on the left hand side and the actual log information on the right hand side. The tree-based filter can be used to reduce the log information. On click on a log information the log messages is shown underneath the header information. The clear log button on the top right hand side can be used to delete log information which is no longer needed.

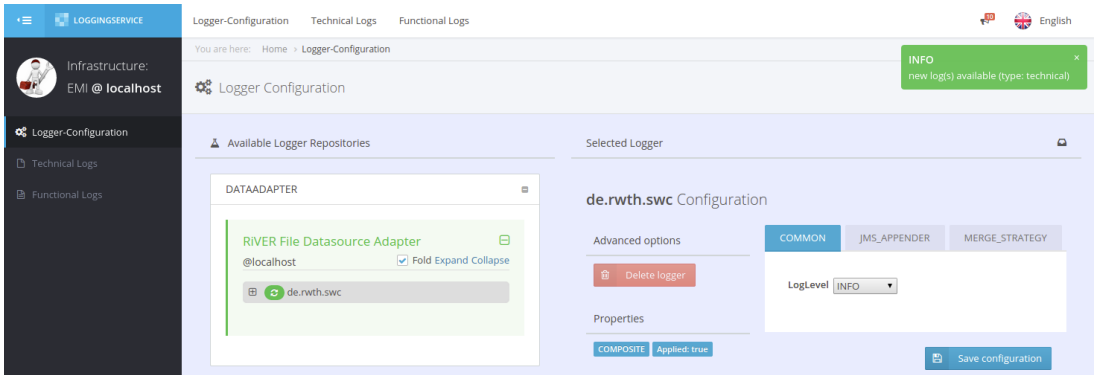


Figure 14.12.: Screenshot of the logger configuration in the GUI of the EMI Logging Service (adapted from [Dör14] p. 68).

Figure 14.12 shows the logger configuration in the GUI of the ELS. The left hand side of the center area again shows a list of all services with a tree of all loggers inside them. The right hand side of the center area shows the configuration of a particular

logger. Each logger can provide configuration parameters. Hence, this view can change from logger to logger. However, most loggers provide the configuration of the log level as shown in the selected tab. On a click on the save configuration button the configuration is sent back to the service and is applied by the specific logger.

14.4.3. EDS - EMI Directory Service

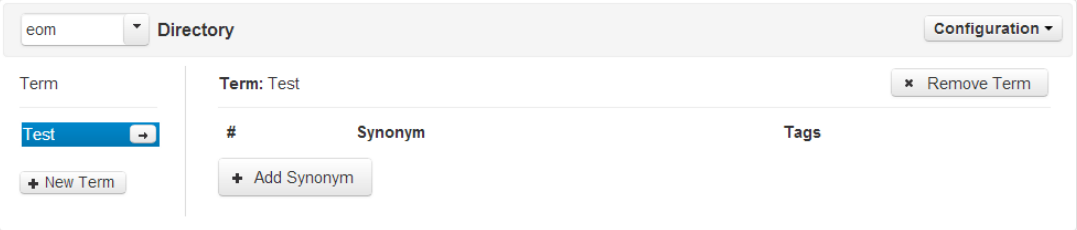


Figure 14.13.: Screenshot of the EMI Directory Service graphical user interface.

Figure 14.13 provides a screenshot of the graphical user interface of the EMI Directory Service. Most importantly, the service supports different directories, which can be selected on the top left. The most common directory is the eom-directory, which holds terms and synonyms for entities of measurements. Below the selection on the left hand side the GUI shows all the terms in the given directory. A new term can be created using the new term button. The right hand side shows all the synonyms for the selected term on the left hand side. Additional synonyms can be added to the term using the add synonym button. This will open a dialog which shows synonyms which are not yet assigned to a term. These can be added to the current term. Additionally, the user can add synonyms via a text field in the dialog in order to add synonyms which are not yet known in the EMI.

14.4.4. ERS - EMI Render Service

The goal of the EMI Render Service is to manage and extend the renderer of a dashboard or analysis tool without the need to redeploy the particular application in a production environment. Therefore, the visualization capabilities of the visualization frontends can easily be extended to satisfy emerging needs. This, again, provides a lot of flexibility to the measurement infrastructure.

Figure 14.14 shows a screenshot of the configuration of a cartesian-chart-renderer in the GUI of the EMI Render Service. Most importantly, the renderer can define its required data structure on the top. This data structure acts as the requested data type for discovering suitable metric kernels via the `DiscoverAPI` from a EUrEKA Registry (see section 5.3.4 for further details). It can also be used in reverse in order to find suitable visualizations (renderer) for a given data type from a metric kernel. Next, the developer of the renderer can provide a list of data representation types. These types can act as additional specific configuration for each data row from a metric kernel in a larger

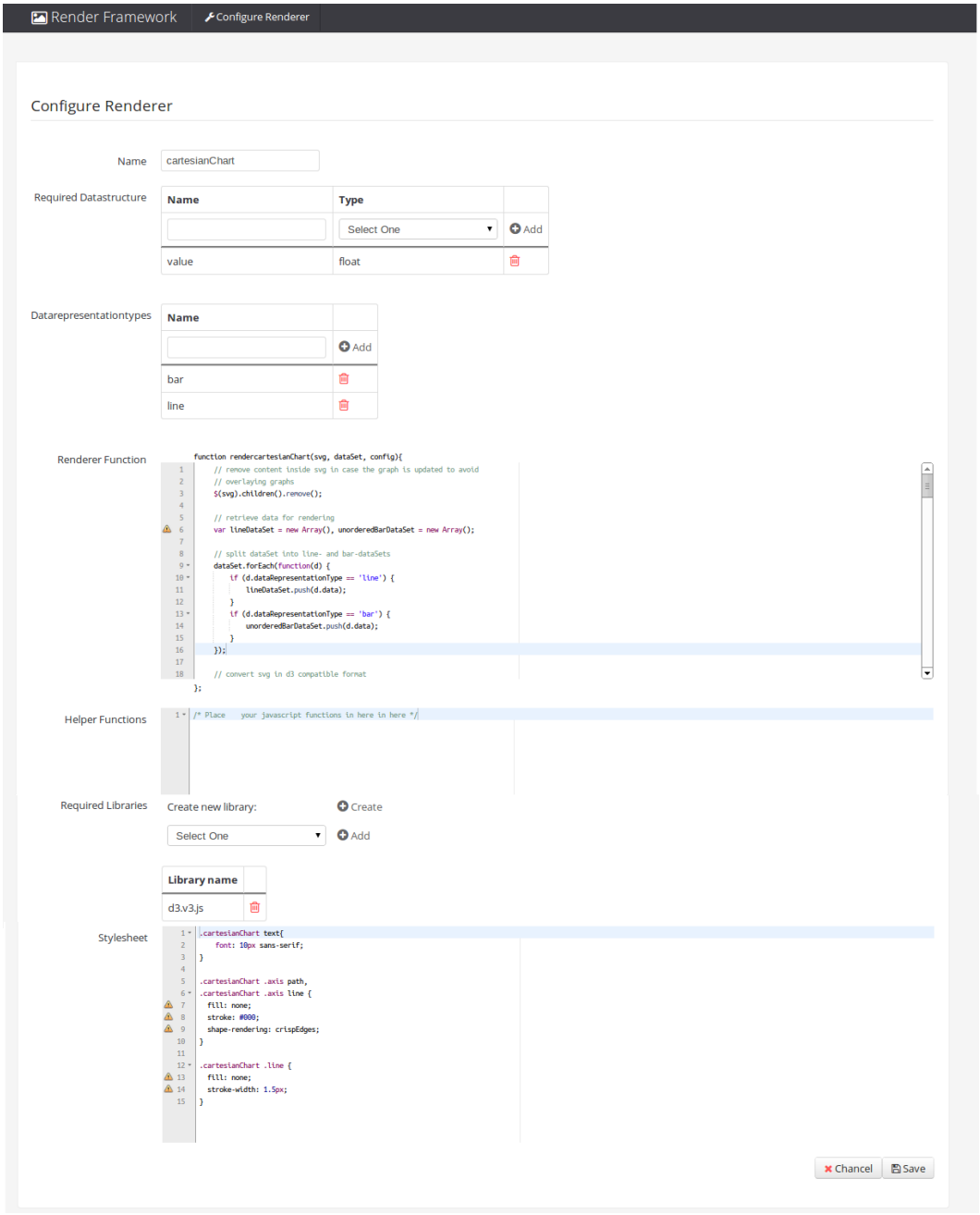


Figure 14.14.: Screenshot of the configuration of a specific renderer in the GUI of the EMI Render Service (adapted from [Röl13] p. 61).

data set. We used a EUrEKA Indicator Wrapper (see section 5.3.7 for further details) to associate the data type as additional payload to each metric kernel configuration and feed it back to the renderer.

The render function in the next field is the actual javascript function which implements the renderer. The function needs to fit to a given signature. Hence, only the function body of the render function can be defined. Furthermore, the developer can add additional helper functions to the renderer configuration which can be use to modularize a large render function.

Typically, renderer use or extend existing rendering and visualization libraries. Therefore, the ERS provides mechanisms to manage and use these libraries. The libraries can be associated with the renderer via the required libraries mechanism shown in the screenshot. The ERS then provides all required libraries for a given set of renderer as a uniform resource which eases the usage of the renderer.

The developer can use the last form field to provide a number of styles for each renderer. These styles define the visual appearance of the rendered graph. The styles can also be accessed uniformly for a given set of renderer.

14.5. EMI Framework

The EMI Framework is a library of enterprise java beans based components which we used to build the EMIs and their services for our field studies. The framework contains components for easy integration of the monitoring client agent, the logging client agent, and the directory service client agent. It also provides the common basis for the communication over the EMDB as well as additional utility classes.

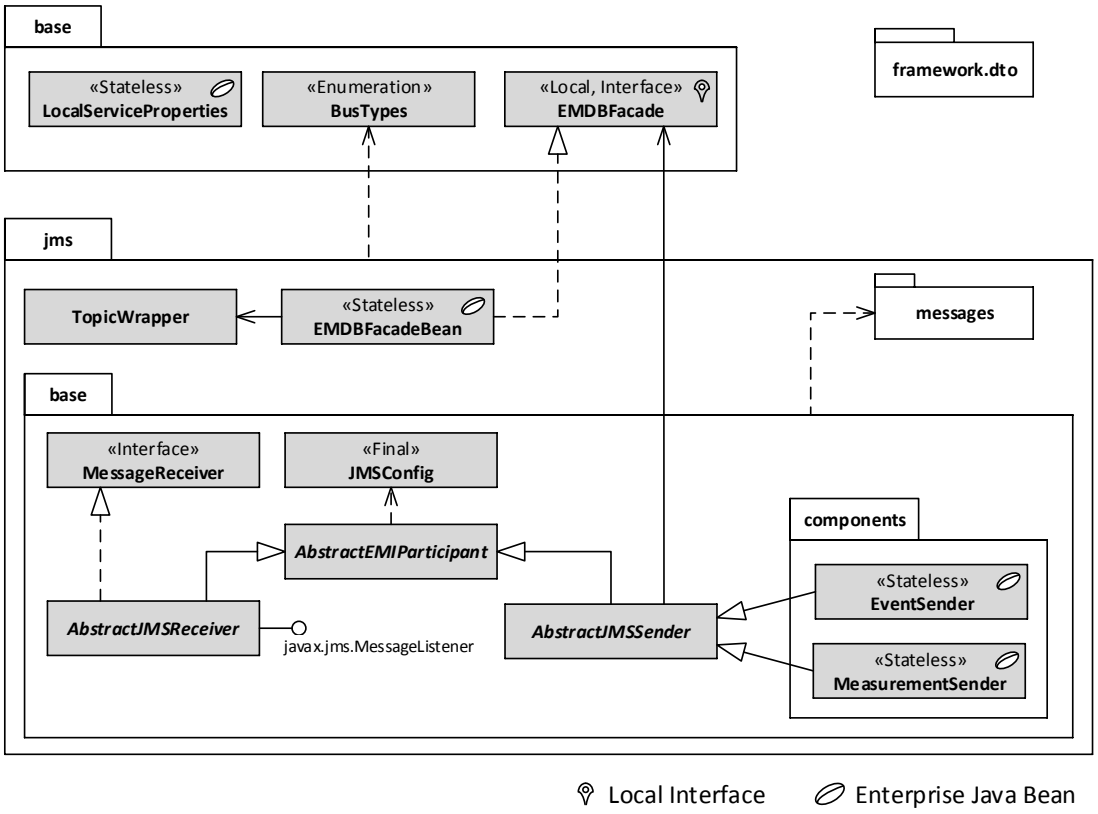


Figure 14.15.: UML package diagram of parts of the common core of the EMI Framework.

Figure 14.15 provides a UML package diagram of the common core of the EMI Framework. The base package contains the `LocalServiceProperties`-class, which provides access to the `emi.properties`-file from each service, as well as the basis for the communication over the EMDB. Most importantly, it contains the `EMDBFacade` local interface, which provides a facade for the message sending on the EMDB.

The EMI Framework uses the java messaging service (JMS) as a basis for the EMDB. Therefore, the `EMDBFacade` local interface is implemented by a stateless enterprise java bean in the jms package. This bean uses a number of `TopicWrapper` instances in order to send messages to the different topics defined in the `BusType`-Enumeration in the base package. Furthermore, the jms package provides additional classes and beans in order to ease the implementation of EMDB-Senders and EMDB-Receivers.

The AbstractJMSReceiver-class provides a basis for all EMDB message receivers. It connects the jms interface javax.jms.MessageListener with our abstraction in the MessageReceiver interface. In order to ease the sending of messages on the EMDB the AbstractJMSSender-class provides a common base for dedicated sender components. The EMI Framework already ships with two sender components for sending event messages and measurement messages to the EMDB via the EventSender and MeasurementSender EJBs. These can easily be injected into the core implementation of data adapter or metric kernel in order to send (standard) measurements and events to the EMDB.

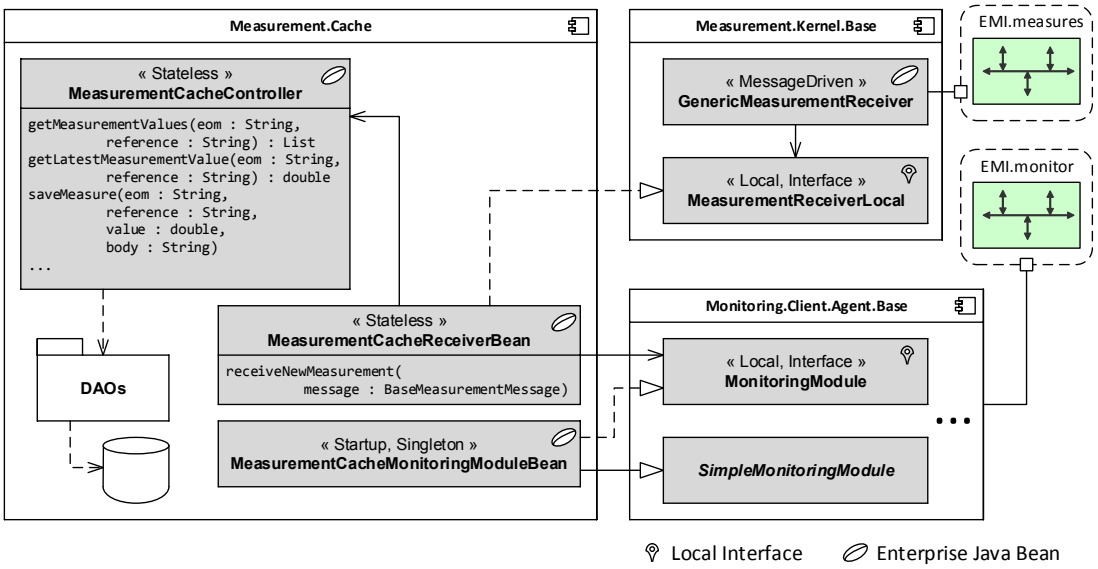


Figure 14.16.: Example for the implementation of a measurement cache using EMI Framework components.

Figure 14.16 provides an example for application of the framework. This example provides a quick glance at the implementation of a measurement cache, which specifically caches measurement messages. The two components from the EMI Framework on the right hand side Measurement.Kernel.Base and Monitoring.Client.Agent.Base further abstract the components and classes in the common core shown before in order to further ease the implementation of standard EMI functionality.

The Measurement.Kernel.Base component implements a message receiver for measurement messages on the EMDB, using the AbstractJMSReceiver class from above (not shown in figure). It then delegates the processing of the message to the MeasurementReceiverLocal interface which is not implemented in the component itself. The Measurement.Cache component on the left hand side implements the interface with the stateless enterprise java bean MeasurementCacheReceiverBean and delegates the processing of the received measurement message to the saveMeasure method of the MeasurementCacheController EJB.

Obviously, the new measurement cache should also be integrated into the EMI

monitoring system. The `Monitoring.Client.Agent.Base` component again provides a local interface, the `MonitoringModule` interface, in order to easily integrate into the EMS. To further ease the implementation the component also supplies a default implementation, the abstract class `SimpleMonitoringModule`, for the interface which is suitable for most cases. The `MeasurementCacheMonitoringModuleBean` simply connects these two. The `SimpleMonitoringModule` then utilizes the `LocalServiceProperties` EJB to get the name, description, and type (not shown in figure).

In order to further ease the usage of the framework we provided additional development support by the means of maven archetypes and test-pattern. Archetypes are a mechanism integrated into the maven build and dependency management system. They can be used to reuse maven projects setups. We provided archetypes[4] for the different types of data adapters and metric kernels. Besides the general project layout our archetypes also included all the dependencies to the required components of the EMI Framework and provided stubs for the core classes. They also included stubs for the different tests defined in the EMI test pattern[5].

[4]See master thesis from Martin Lang [Lan14] for further details.
[5]See bachelor thesis from Marco Moscher [Mos14] for further details.

15
Lessons Learned and Discussion

The previous two chapters provided selected field studies as well as descriptions on some of our EMI specific tooling which we used in the field studies. Each of the three field studies already provided discussion sections. However, we like to condense the lessons learned from all our field studies and provide some additional discussion on our concepts in this section.

We did not perform specific case studies or experiments in order to evaluate certain aspects of our approach. Our resources in the university are limited. Case studies and experiments can only be performed with (master and bachelor) students. Our approach, however, should not just provide benefits in situations in which students (with very few industrial experience) are first faced with our approach. It should provide benefits in an industrial environment with experienced developers, architects, and requirements experts. Thus, we would have needed to conduct the case studies with them. These resources are very valuable and their time is very costly. Furthermore, it is near impossible to control all important variables in these experiments. Thus, the outcome would be very questionable and most likely not applicable in other environments and scenarios. Therefore, this evaluation section is based on our experience of the application of our approach in the field. Even though our experiences are very specific we belief they provide a good basis for the evaluation because they are based on real world environments.

We separate this chapter into five sections. Each of these sections will address one important aspect for our lessons learned and further discuss our approach. The first section 15.1 will discuss the most requested aspects from our cooperation partners: security. After that we will reflect on the core aspect for the design of our approach: Flexibility in section 15.2. Another goal of our engineering approach was to build "usable metric systems". Hence, the following two sections section 15.3 and section 15.4 will discuss two important aspects of usability: Ease-of-use and effectiveness (of the process). Last, section 15.5 will discuss the efficiency of our engineering approach because monetary and time aspects should obviously be considered when engineering a metric system.

15.1. Security

Security was the most requested feature from most of our cooperation partners. They particularly wanted to secure the communication on the EMDB in order to avoid that unauthorized persons read confidential data. However, we did intentionally not include a mechanism into our reference architecture to secure the communication. It would, of

course, be possible to secure the internal communication inside the EMI. But securing the communication requires a lot of additional effort. For example encrypting and decrypting messages on message send and receive could be used to further secure the communication in an EMI. However, this would raise the complexity of the whole EMI and would have huge impacts on some of the important operation services like the message cache. Additionally, this mechanism can be hacked for example by man-in-the-middle attacks. Hence, the mechanism needs to be even more sophisticated like two way encryption which increases complexity and reduces performance even further. Furthermore, there are easier ways to (partially) secure an EMI, which is sufficient for most of the attack scenarios.

We utilized a number of simple means to partially secure the EMIs by using external mechanisms and some restrictions on the architecture. We emphasize that this approach is only able to partially secure the EMI because the internal communication, inside the EMDB for example, is left untouched. Therefore, an intruder that breaks into the system is able to read all the communication in the EMI by connecting to the message buses. However, we will argue that an intruder who is able to break into the system like that would also be able to easily break other security mechanisms. Therefore, we decided to make intrusion as hard as possible using standard mechanisms, which should hold back the vast majority of possible intruders.

Our security concept for sensitive data, for example in our SSE Lab field study (see section 13.2), was three folded. First, we made sure that read and write operations on all endpoints in the EMI are exclusive. Therefore, intruders can not simply use a data write endpoint for read and vice versa. Secondly, we located the core of the EMI in a secured network environment with a firewall between the two networks. We only grated dedicated services (data adapters and dashboard frontends) access to the EMDB and the indicator access APIs of the metric kernels. For the SSE Lab field study we also wrapped the access to the dashboard frontend to secure it even further. Thirdly, we used the security mechanisms from our Java Enterprise Edition application servers to secure the services which face the users. Breaking even one of these security barriers is very hard! Thus, someone who is able to break these would also be able to break the possible security mechanisms inside an EMI. For example by injecting false signatures in a signing mechanisms for messages on the EMDB via a man-in-the-middle-attack or replacement of the signing mechanism. Therefore we did not integrate dedicated security support into the EMI reference architecture. This security concept was sufficient for all our cooperation partners and all our field studies.

One of the lessons learned, however, is to address security very early in the process and have dedicated scenarios and designs available. For example by using some or all of the concepts presented above. Some of these will have impact on the server infrastructure or network and firewall settings. The operation department will need to know this information in advance in order to setup the correct production environment. Addressing security aspects early in the requirements phase of a software development project, however, is a common best practice and should always be applied!

15.2. Flexibility

In our field studies technical flexibility and decoupling of all the different services in our reference architecture allowed us to choose the optimal technology for metric kernels and data adapters on a very fine scale. We could choose the implementation technology for each metric kernel independently and most importantly we were able to freely select the most suitable database technology for each kernel. For our field studies this was particularly important because we needed to deal with very heterogeneous data. Hence, we selected among a brought variety of data bases; ranging from relational databases for some metric kernels over graph databases to document stores and noSQL databases for others. The third field study on the RiVER tool in section 13.3 shows the tremendous impact this can have. Our process model supported flexibility by dedicated designs for each metric kernel, data adapter, and visualization frontend. In our field studies the iterative and incremental nature of the process was very suitable to address the given challenges.

We also reflected flexibility in our tools. Most of the tools were developed in isolation from the rest and then seamlessly integrated to address the requirements of a given metric system and field study. The EMI Framework allowed us to easily implement new metric kernels and data adapters and supported the development of our visualization frontends (like MeDIC Dashboard, SCREEN, and the RiVER frontend). The ready-to-use EMI services (EMS, EDS, ERS, ELS, and EMC) together with the EUrEKA services provided a solid base for each EMI that we built, which again shows the flexibility of the solutions and our engineering approach.

15.3. Ease-of-Use

The discussion on ease-of-use is two folded because we need to discuss it separately for the process model and the reference architecture. Therefore, we further separate this section into dedicated subsections.

15.3.1. Ease-of-Use of the Reference Architecture

The ease-of-use of the reference architecture can be evaluated by our effort that we need to spend on training of new developers. Most of the students who worked on tools and field studies needed to learn the reference architecture in order to apply the concepts. We supplied our papers, specific documentation, and our projects as training material. Most of the developers grasped the concepts of the reference architecture within a few days and were able to use its concepts within a week. Furthermore, they all mentioned the soundness of the concepts. They reported that, especially the concepts of data adapters, metric kernels, and the EMDB were very easy to understand. The concepts on top of the metric kernels and interconnection between the metric kernels and visualization frontends using the EUrEKA concepts were harder to grasp. We believe one of the reasons for this is the lower structural complexity of the bottom concepts surrounding

the EMDB; they are all simply connected via the EMDB. The services in the top part of the reference architecture do not utilize a single publish-subscribe infrastructure but are based on a classical service oriented architecture. Hence, the structural complexity is higher and especially the implicit coupling mechanism is harder to apply in real world scenarios. However, we already discussed the pros and cons of this type of infrastructure in section 5.1.3 and still believe it was the right choice. Also, once the concepts were understood, the students and other developers could easily use all the concepts without any problems, which again shows the ease-of-use of our reference architecture.

15.3.2. Ease-of-Use of the Process Model

We developed the process model together with the field studies and refined it with each one. Therefore, each field study benefited from the previous one. Especially the artifacts and templates were very useful for the later projects. The design documents and all the surrounding design activities proved to be very easy to use. Furthermore, everybody who needed to adhere to the process and use the artifacts immediately saw their benefits. We had some discussions, especially with developers, over the perceived document heavy and design focused process. But they also quickly saw the benefits in the solid design foundations once they started implementing dedicated solutions. Thus, we believe this indicates the very easy use of the activities and artifacts of our process model.

15.4. Effectiveness

In this section we like to discuss the effectiveness of our engineering approach. Again we like to separate the discussion for our process model and our reference architecture because they address different aspects which a discussion on effectiveness needs to reflect.

15.4.1. Effectiveness of the Reference Architecture

All the tools and services that we built in our field studies show the effectiveness of our reference architecture. All of them are able to provide the measurement infrastructures needed for the particular metric system. Furthermore, the field studies also show the flexibility and reusability of the services. We were easily able to reuse metric kernels and data adapters. We could also easily tailor some of them to include additional functionality or extend them with additional services. Maintenance of the services also proved to be very easy due to their micro service nature. They could be tested and bugs reproduced in isolation, which eased bug fixing tremendously. Additionally, the reference architecture proved to be understandable as developers and architects could easily use it.

The reference architecture also needed to pass a formal appraisal by the architecture management board at one of our cooperation partners. We provided our training material and additional insides into the reference architecture before the actual hearing which feed some initial discussions. During the hearing we presented the core principles and concepts to the board. After that we answered some additional questions on technical details. The reference architecture passed the appraisal with flying colors and the board

emphasized the sound concepts and design of the reference architecture. This again underlines the effectiveness and soundness of the reference architecture.

15.4.2. Effectiveness of the Process Model

Our field studies, especially those with large cooperation partners, showed the effectiveness of our process model. The heavy focus on requirements and design in the early phases of the process model helped to understand and pin-point the needs from the metric customers. Especially the prototyping activities and related evaluation with metric customers proved to be very useful. They typically resulted in detailed discussions and showed additional needs of the metric customers which would otherwise be lost. We also experienced complete changes of proposed monitors which did not suite the needs of the metric customers. Fortunately, we noticed these in the prototyping phase and not after implementation or staging.

Again, the templates and artifacts proved to be a very good means for communication and we were quickly able to design suitable solutions for the needs of the metric customers. Especially the design document proved to effectively communicate the design among the different stakeholders and provided solid foundations for the implementation by the developers.

15.5. Efficiency

Efficiency discussions, especially for an engineering approach, are always hard because no two projects are identical. Additionally, we can only discuss the efficiency for complete projects and not separately for the process model and the reference architecture. Luckily, one of our cooperation partners did develop an excel-based project management dashboard prior to our EMI field study. Therefore, we can roughly compare the effort of the two projects (also see [VLJ13]). The numbers are taken from time sheds and discussions with our external colleagues. Unfortunately not all activities are documented as desired. Hence, we needed to approximate some the numbers.

The development and adaption of the excel-based dashboard required approximately 78 person days in three months. From this effort 30 person days were required by an external consultant, 40 person days were required for two metric experts and 8 person days were required for a developer. The maintenance of this solutions over the next 3/4 of a year required approximately 97.5 person days. 52.5 person days from this were again required by two metric experts. The remaining 45 person days were required for a dedicated developer who was working part-time on the project.

The effort for our first EMI increment in the same environment was approximately 64 person days which were also spend in three months. Two metric experts required approximately 19 person days. A dedicated architect required 15 person days and the remaining 30 person days were spent on developers. Over the next 6 months we only spent 8 person days on maintenance tasks for the services from this increment. Before the initial development we approximately spent additional 20 person days on initial

requirements gathering, interviews, and pre-project tasks.

The effort spent on the EMI solution, especially the maintenance effort, is much lower then the effort for the initial excel-based solution, which was developed without a dedicated development process. Additionally, most of the effort was spent on lower cost resources (developers instead of metric experts and external consultants).

However, the EMI was not used in a production environment with all projects accessing and working with the system and the scope was much smaller. Hence, as expected in a field study, the results are rather hard to compare. However, especially the initial effort spent on the requirements and the design seams to pay of in the long run. Also, the initial effort for the EMI solution following our development process was lower than expected and resources were far better utilized, which shows the efficiency of our engineering approach.

This concludes the evaluation, tooling support, and lessons learned of our metric systems engineering approach MeDIC. The next part will briefly discuss future work and conclude this thesis.

Part V.

Conclusion and Future Work

16
Conclusion and Future Work

We finish this thesis with three important further fields of studies in the following future work section. The first aspect discusses data quality as an important aspect for measurement messages in the reference architecture. The second one discusses missing explicit tool support for the process model. The third one then discusses further evaluation required for the approach. The conclusion in the final section then summarizes and concludes the thesis.

16.1. Future Work

The most important future work is to include data quality explicitly in the measurement messages and different services in the reference architecture. During the discussion about success factors for metric programs in section 1.2.1 we already discussed the importance of data quality. In the current design of our reference architecture data quality for measurement messages can only be handled implicitly. However, in line with the related work our field studies also showed the importance of communicating data quality. It provides a lot of transparency to the measurement customer who analyses specific measurements in a visualization. It needs to be taken into consideration when interpreting visualizations on a dashboard. Hence, the measurement part of the reference architecture should provide a mechanism to communicate the data quality of the measurement explicitly. Furthermore, the metric kernels need to provide the data quality for their metrics in order to show it in the dashboards and analysis tools. Data quality problems due to errors in some component in the EMI also need to be reported to the monitoring and logging system.

Another future work is a more refined tool support for the different steps in the development process model. We used rich text editors to write the artifacts and drawing tools for our diagrams. However, this can be supported by specialized tools. These tools could also support the different steps in the process model with notifications and artifact handling as well as discussions and decision support. Furthermore, explicit models of the development artifacts could be used to support model-driven engineering of some of the components. For example the measurement senders and receivers as well as measurement messages are optimal candidates for such an approach. They are easy to model, the interfaces are always identical and heterogeneous technology environments require different technical implementations for the same measurement message.

Finally, our approach requires further evaluation. We performed a lot of field studies in order to evaluate our metric systems engineering approach; we just selected three of them for this thesis. However, the approach still requires further evaluation. Most importantly long term field studies are missing. Like in many other engineering approaches that originate from university research. Without these long term field studies our discussion on the success of the approach can only be based on indicators from our (short term) field studies. Furthermore, dedicated experiments or case studies could be performed to evaluate specific parts of our approach. However, as discussed in section 13 these need to be performed in an industrial context in order to provide reasonable results for our approach[1]. This will make them very expensive and hard to perform.

16.2. Conclusion

This thesis presented our metric systems engineering approach MeDIC. The goal of MeDIC was to support the engineering of flexible, information need driven, and usable metric systems. We identified these aspects as the main challenges, which metric systems need to face in industrial environments. Furthermore, our goal was to fill the "development-gap" between the metric portfolio and its associated metric management activities on the one hand and the measurement infrastructure on the other hand.

We first thoroughly investigated the challenges and requirements for metric systems engineering based on our experience and related work in the literature. From this we further investigated and discussed two related approaches. The requirements from this first section guided the definition and development on our two main contributions: our metric systems engineering process model and our reference architecture for enterprise measurement infrastructures (EMIs).

We then formalized certain aspects which are only briefly described in the first chapter. Most importantly we defined and decomposed metric systems into their two parts: metric portfolio and measurement infrastructure. We then provided a more formal introduction to our ideas for metric reuse and metric variability. Last but not least we defined a formalism for metric system dynamics, which provides further requirements for the reference architecture and certain activities in the metric systems engineering process model. The formalism provides a valuable framework for the formal discussion of metrics on the conceptual level in the metric portfolio before the actual (technical) solutions are designed. Using this framework we can discuss termination of the calculation of the metrics before integrating the different parts in the measurement infrastructure.

The reference architecture part of this thesis first refined the requirements from the challenges and top level requirements. From these we motivated our polylithic micro service-based design foundations for the reference architecture. The first part of the actual reference architecture provided a reference architecture for the logical view on measurement infrastructures. This already reflected the four important layers and provided the concept of *metric applications* in order to further structure large sets of

[1] The typical experiments with students will not provide meaningful information about the application of the approach in an industrial setting with experienced developers.

components in these layers. These metric applications also guided the development increments proposed in our process model later in the thesis. Additionally, we further classified the different systems in a physical system view which provided the foundations for the technical reference architecture.

We then presented the technical core of our reference architecture. We proposed to design the central metric related services in three domain layers: *Measurement*, *Calculation and Storage*, and *Visualization* and integrate the services by two dedicated integration layers: *Data Transport and Integration* as well as *Calculation Access*. This layered architecture of the measurement system core is then flanked by the operation layer which contains services that aid and enable typical operation tasks of an EMI. The two integration layers provide the basis for the flexibility and reuse potential of the measurement infrastructure and its services. The technical reference architecture also supports the design of the core services in the domain layers with dedicated pattern and ready to use designs. Our field studies in the evaluation section our very positive experience with using the reference architecture in industrial environments and on real world projects. We belief the reference architecture is very useful and easy to use. It has a high understandability and helps to effectively and efficiently design and build metric systems in industrial contexts.

Additionally, we extended our formalism from the initial part in order to reflect the concepts proposed in the reference architecture. This provided the basis for some design restrictions on the services in the reference architecture; like the "stability" requirement for metric kernels. Furthermore, the formalism contributes the basis for proving the correctness of the message-based implementation of a given metric as well as the basis for investigating the termination of a given EMI. We also provided a very detailed example for the application of the formalism in order to better understand the concepts. The example also showed the practical relevance and applicability of the formalism.

The next part then discussed our metric systems engineering process model. The process model further bridges the development gap between the metric portfolio and the measurement infrastructure. We also included activities to support the operation of the measurement infrastructure which is often neglected. We designed the core of the process model to follow the typical software developments tasks in a PDCA-oriented, circular, and incremental manner. Thus, our process model is based on four dedicated phases: *Conception*, *Design*, *Construction*, and *Operation*. We performed a very thorough analysis of the roles involved in the metric development process and listed their needs and responsibilities as well as their involvement in the different phases of our process model. From there we started to define the activities in each of the phases.

We provided very fine grained activities in the conception phase with a lot of details because we believe it is important to get the requirements right before starting the construction of the software solution. We further included dedicated prototyping activities in this phase in order to reiterate on the solution with the metric customer before starting with the actual software design and implementation. This is particularly important in large organizations because the larger the organization the harder it is to change the metric system. We also included the increment planning into the conception phase. Each

development increment can then be addressed separately in the following phases.

The definition of the design phase is more open then the definition of the construction phase. We only included the obvious activities and left the actual activities open for the instantiation of the process model because the metric experts and architects need to be able to flexibly adapt the process model to their situation. The main artifact that is defined in this phase is the design document which provides the basis for the further construction activities (implementation and staging including test). The evaluations in chapter IV show the strengths and benefits of the design document as well as the interconnection between process model and reference architecture in this phase.

The construction and operation phase are even more specific to the actual development environment. Therefore we only included best practices for the staging environments for measurement infrastructures in the description of the construction phase. For the operation phase we provided best practices for handling typical errors and exceptions. For each of these we provided ways to identify the error or failure as well as (multiple) resolutions for recovering or avoiding it. This part further interconnects the process model and the reference architecture because some of the solutions and detection mechanisms require dedicated services in the infrastructure which are proposed in our reference architecture. Again, the field studies and tools show the strengths and applicability of our concepts in industrial and research environments.

The different facets of our engineering approach, especially the reference architecture, were well received by the research community. Our publications served as a valuable base for discussions together with the community on different aspects which helped to enhance and further streamline our approach. Furthermore, we were able to spark various research projects based on our work.

We conclude that our metric systems engineering approach MeDIC is able to successfully bridge the development gap between the metric portfolios and measurement infrastructures with a dedicated metric systems engineering process model and reference architecture for enterprise measurement infrastructures in industrial environments. MeDIC leads to more flexible and usable metric systems, which are easily able to adapt to inevitable changes in their environment. Thus, MeDIC provides the basis for the long term success of metric programs.

Part VI.

Appendix

A
Symbol Lists

In section 2.3 we presented a formalism for metric system dynamics. This formalism uses a broad variety of symbols for the different parts of the formalism.

A.1. Symbols used in the Foundation Formalism

The following list provides a description for most of the symbols used in our formalism for metric system dynamics in section 2.3. This can be used as a guide when reading the section.

$\overline{n}$ — Abbreviation for the set $\{1, \ldots, n\}$.

T — The type for a measurement data set.

t — A measurement data type component.

d — A measurement data set. We also refer to this simply as the measurement data.

D — A set of measurement data.

kv — A key-value pair. These make up the measurement data sets $d = \{kv_1, \ldots, kv_n\}$.

$type(d)$ — The type function for measurement data. Returns the type of the measurement data d.

$type'(kv)$ — The type function for a key-value pair.

$\mathcal{T}$ — The type for a measurement.

$\mathcal{M}$ — A measurement. A measurement is represented by a tuple: (d, M_{id}, eom)

$type(\mathcal{M})$ — Type function for a measurement. Returns the type of the measurement as a tuple: $(T(d), M_{id})$.

M_{id} — A measurement identification. Our formalism assumes a name-space concept on these using "." as a delimiter.

eom — An identifier for an entity of measurement.

$\prec$ — The *compatibility* relation between a measurement (data) and a measurement (data) type.

$guard(\mathcal{M})$ — The guard function of a measurement consumer.

$\texttt{guardGen}(\mathcal{T})$ — The generator function for type specific guard functions. This returns a guard function that accepts measurements that are compatible to the given measurement type.

$\mathbb{T}$ — A set of measurement types. This is typically used to represent the set of types used to generate the guard function for a measurement consumer that accepts all measurements that satisfy one of the types in the type set. This is, hence, sometimes called the *accepted type set*.

$\mathbb{M}$ — A set of measurement.

$\models$ — The *satisfies* relation between a measurement set and a measurement type set.

MS — A metric system.

$\mathfrak{M}$ — A metric. This is typically used for derived metrics.

C — A measurement consumer.

P — A measurement producer.

$E_P(EOM)$ — The calculation function for the output entity of measurement of a measurement producer P.

$produce_P(\mathbb{M})$ — The production function for the measurement producer P.

$produce_{\mathfrak{M}}(\mathbb{M})$ — The production function of the derived metric $\mathfrak{M}$.

$\rightleftarrows$ — The feed relation between two metrics.

$\mathfrak{C}$ — A calculation chain of derived metrics.

A.2. Symbols used in the Reference Architecture Formalism

The following list provides descriptions for most of the symbols used in the formalism for our MeDIC reference Architecture in section 7.2.

$f_1||f_2$ — ||-Operator to formalize that a function f_1 is executed before a function f_2.

$\mathtt{m}$ — A measurement message.

$\mathtt{M}$ — A set of measurement messages.

ts — A timestamp in a measurement message.

L — The latency of a measurement message.

L_{avg} — The average latency of a data adapter

Raw — Set of raw data in a data provider.

$\mathfrak{S}$ — A arbitrary service in an EMI.

$\mathfrak{A}$ — A data adapter in an EMI.

$\mathtt{M}_{\mathfrak{A}}$ — The set of measurement messages produced by the data adapter $\mathfrak{A}$.

$\mathfrak{K}$ — A metric kernel in an EMI.

$\mathbb{T}_{\mathfrak{K}}$ — The accepted types set of the metric kernel $\mathfrak{K}$.

$store$ — The internal data store of a metric kernel.

$persist$ — The storage function of a metric kernel.

$\mathcal{I}$ — An indicator access API on a metric kernel.

$\mathtt{view}_{\mathcal{I}}$ — A view transformation function for an indicator access API $\mathfrak{I}$.

$produce_{\mathfrak{K}}$ — The production function of the metric kernel $\mathfrak{K}$.

$\widehat{produce}_P$ — The robust production function of the measurement producer P.

$\widehat{produce}_{\mathfrak{M}}$ — The robust production function of the metric kernel $\mathfrak{M}$.

$\mathtt{M}^0$ — The initial data processing step of an EMI after adapting the data from the data provider.

$\mathtt{M}^*$ — The outer data processing hull of an EMI.

B

Process Guides, Checklists, and Document Descriptions for the Process Model

This chapter contains accumulated guidelines, checklists, and document descriptions from our experience of using the process model in our field studies.

B.1. Conception Phase

This section provides additional information and best practices for the conception phase of our process model discussed in section 10.

B.1.1. Information Need Gathering – Guidelines for the Execution

The following two subsections contain some accumulated guidelines for the execution of the two information need gathering techniques discussed in section 10.1.

Best Practices

It is often a good idea to start with a question like: "What lack of metrics/monitoring annoyes you the most in your everyday work?" Because this gives a good insight into the metric customers daily routine and usually provides a good basis of important information needs. It also helps to get the discussion going because everybody can easily answer this question. It may also provide a good insight into the processes this metric customer is working in if this questions reveals different areas of annoyances.

Also include a question like: "What metrics/monitoring tools works great in your everyday work?" The metric experts need to make sure to also focus on keeping these in tact while introducing new monitoring tools and metrics.

During an interview or an initial workshop session the metric experts can point the metric customers to the four important areas of project management: costs, scope, quality, and time. This is of course important when interviewing (project) managers but these areas are also important for other roles - especially cross-cutting roles like configuration mangers, test managers, or quality assurance specialists.

Document information needs as questions. This maybe obvious, but sometimes it is quiet hard in a specific situation to pin point the exact question that should be answered. Still it is important to do this to avoids vague information need design. However, we realized, that it is not important to specify goals exactly like specified in GQM (5

characteristics). Event though all the characteristics are important, specifying goals in this exact manner in a workshop or interview puts too much focus on details.

Workshops

- ✓ At least two metric experts need to participate in each workshop. They act as moderators and try to keep track of all the goals and questions that the participants mention.
- ✓ The group of metric customers should be homogeneous (same role in the company, same expertise, or same problems) this is to avoid to many "basic" discussions in the workshop.
- ✓ The workshops should be organized as group sessions which not exceed 1.5 hours!
- ✓ The focus of the workshop and each session needs to be stated before starting each workshop session.
- ✓ The first workshop session should include a small introduction into GQM, the goal of the workshop, and the approach.
- ✓ In general the approach should be to first try to get a broad set of goals and then focus on the important ones and get deep.
- ✓ First start with letting the metric customers identifying their rough goals, then refine them towards simple questions which can be answered by metrics (GQM approach)
- ✓ The metric experts (moderators) need to keep the metric customers focused (depending on the focus of the workshop at hand!).
- ✓ The metric experts should also summarize the results every 15 too 20 minutes to give the metric customers a break and point them to important areas that are not covered yet.
- ✓ On the start of every workshop in a series the metric experts should present distilled results from the previous workshop to get the group going.

Interviews

- ✓ The metric experts should prepare a questionnaire for each interview to organize the interview. The questionnaire should contain:
  - Questions on the role of the metric customer. This is used for the later integration of the interview results.
  - Specific questions on the information need of the customer.

- ✓ The interview should (like the workshop) start with a small introduction of GQM and the goals of the interview.
- ✓ If the goal of the interview is to evaluate the current metric-system the metric experts should include a question asking about the metric/monitors that are never used. This will provide the metric experts with a set of metrics/monitors that can either be removed or require more advertisement.

B.1.2. Plan Increment – Guidelines for Coherent Increments

These guidelines are derived from the components that need to be altered during the design and construction phases. Roughly speaking an increment should either change only small parts in many different components on different layers or change large parts which should be limited to only a few components.

- ✓ Information needs for the increment are related to one group of metric customers
- ✓ Information needs for the increment belong to the same category (e.g. risk management, error analysis, or process compliance)
- ✓ Prototypes for the information needs for the increment use the same visualization
- ✓ Information needs can be answered by a similar type of metrics (e.g. counting metric)
- ✓ The information to provide answers for the information needs is derived from the same data provider(s).

B.2. Design Phase

This section provides additional information, guidelines, checklists, and document descriptions for the design phase of our metric systems engineering process model.

B.2.1. Services Reuse Decision Aid – Checklist

This checklist helps to determine if the requirements in the increment plan can be realized by reusing existing metric services. It also helps to identify potential changes if existing metric services are reused. This of course requires a precise and up-to-date documentation of the existing metric services.

Dashboard Application

- ✓ Check for new or changed interactions among the different monitors on the dashboard
- ✓ Check for new or changed management functions for the monitors like move, resize, reuse, and most importantly (because difficult and maybe not addressed) access rights
- ✓ Check for new or changed requirements on dashboard management (reuse, sharing, and again: access rights)
- ✓ Check if the dashboard application can be used in the environment (web vs. desktop vs. mobile app) of this increment

Visualization

- ✓ Check for new or changed diagram types
- ✓ Check for additional or changed data in existing diagrams
- ✓ Check for new or changed interactions (select, move over, zoom)
- ✓ Is the existing visualization infrastructure technically capable of delivering the required functionality (ex. environment change)

Metric Kernel

- ✓ Check for new or changed calculation functions (this also includes new aggregation levels or aggregation functions)
- ✓ Check for new or changed message types (this requires a design of the integration)
- ✓ Check for new or changed data access APIs (i.e. new or changed data for visualizations)

Data Adapter

All types of data adapters

- ✓ Check for new or changed data required for a metric kernel
- ✓ Check for new or changed data in the data provider
- ✓ Check for new or changed configuration requirements to the data provider

Push-Forward

- ✓ Check for new or changed plug-in API at the data provider
- ✓ Check for new or changed data required at the data gateway
- ✓ Check if new adapted data provider can reuse an existing data gateway. If so does the existing gateway needs to be changed (additional or fewer data fields from the new data provider)?

Pull-Forward

- ✓ Check for new or changed timing required for the data
- ✓ Check for new or changed API access of the data provider

Invoke-Push and Forced-Dump

- ✓ Check for new or changed API access of the data provider
- ✓ Check for new or updated event data / data in the dump command

Operation Service

- ✓ Check for new or changed logging and monitoring requirements
- ✓ Check for new or changed common services required in the concrete EMI

B.2.2. Design Guides for EMI Services

The following subsections contain design guides for the key EMI services: Data Adapters, Metric Kernels, and Visualizations.

Design Guide for Data Adapters

A data adapters connect the EMI to the data providers. Inevitably, the data providers and adapted data will change over time! Therefore the design of the data adapter should adhere to the following best practices:

- ✓ The adapter should be flexible to a certain degree to address these changes
- ✓ Transformation of data should follow the strategy pattern with dedicated strategy for each transformation
- ✓ Data adapters should avoid overly complex transformations and calculations. This is the job of metric kernels.

Design Guides for Metric Kernels

The metric kernels are the heart of a metric application. Of course the metric kernel needs to be able to provide the metrics required to visualize the monitors that answer the questions of the metric customers included in this iteration. The design of the metric kernel needs to stick to the reference architecture, however, the specific parts provided in the reference architecture need to be specified in more detail; including a database schema for the data storage. This turned out to be quiet useful because metric experts typically can understand database schema.

Due to the design of the reference architecture (specifically the loose coupling via the EMDB) additional non functional requirements for the design of the metric kernel arise (see section 7.2.4 for justification):

- ✓ The calculation and data storage need to be robust against receiving a single message multiple times (the result must not change!)
- ✓ The calculation and data storage must also allow updates (corrections) of old data.
- ✓ The design should generally try to avoid to store aggregated values because it drastically increases the complexity of the controller of the metric kernel (decide when and how to update what data).

Design Guide for Visualizations

The visualizations provide the monitors for the metric customers based on the indicators provided by the metric kernels. The design of the visualization frontends should adhere to the following best practices:

- ✓ Like data adapters visualizations should avoid complex transformations and calculations. This is the job of the metric kernel.
- ✓ Design visualizations to be reusable for different types of (compatible) data
- ✓ Build visualizations in a way that they provide additional information on-mouse-over (if possible)
- ✓ How to handle inconsistent data and how to act on the exceptions listed above.

B.2.3. Design Guide for Test Selection and Test Stage Description

Like all the other parts of a metric system tests need to be well defined. Test selection criterion helps to systematically define test cases. From our experience equivalence-class-based test selection provides good results. An EMI is a collection of loosely coupled services. Therefore, the architect can choose among different integration test strategies. We recommend to test from inner to outer components and integrate from the small components to the big once. This is also strongly coupled with the staging strategy (see section 12.1).

The test section should contain tests for all exceptions defined in the design document (see section B.3.2). Furthermore, for each service it should contain test for the different test (and staging) levels. The following subsections provide additional guides on the design of the tests for the specific levels[1]. Each subsection starts with the description of the test goal for the specific level and then lists the best practices for the test design.

Design Guide for Unit and Module Tests

Goal: Test a single unit (component) in a service

- Units are typically single classes or small modules
- Use standard xUnit Frameworks for this + Mocking Frameworks to isolate the unit from the rest

Design Guide for Component Integration Tests

Goal: Test the integration between components inside a single service

- Integration strategy: Single component at a time then integrate: from inside to outside
- Solid testing requires well cut components which are designed for test
- Requires the components to run in their native environment.
- May require additional effort to mock other services
- Component integration tests show configuration errors very early

[1] We also recommend the bachelor thesis from Marco Moscher who defined test pattern and test templates for these different levels [Mos14].

Design Guide for Service Integration Tests

Goal: Test the integration between services of an EMI

- Integration strategy: Single service at a time then integrate: first adapter, then metric kernels, then visualization
- Requires the metric services to run in their native environment
- May require additional effort to mock other services
- Requires additional effort for test drivers and mocks of data providers
- Check data adapters via message cache
- Drive metric kernel tests via message gateway

Design Guide for Application Integration Tests

Goal: Test the integration between the EMI and the data provider

- Integration strategy: A application and scenario a time
- May require mocks and additional test drivers for isolated test of invoke-push adapter pattern
- Check via message cache

Design Guide for EMI System Tests

Goal: Test the complete EMI with all data providers in an environment similar to production

- End-user test of the complete metric system
- Test is driven from the front-ends and the data provider
- Result check only accesses the front-ends

B.3. The Design Document

This is the core artifact that is created in the design services and integration and design test activity. The fine design of the services provides a very good basis for the service documentation. Even though we describe it as one document it can be cut into several documents. The one document can get large but several documents can become inconsistent more easily. After evaluation the document is handed over to the developers in the construction phase. The following subsections describe the content and goal of the sections in the design document. Each of the following subsections mirrors a subsection of the `Design Document`

B.3.1. Rough Design of the Complete Metric Application

This rough design should contain all services identified as well as a rough overview of the integration. Typically this section starts with an overview diagram of the metric application. We used slightly augmented UML deployment diagram for this (add EAI notations for buses and messages and add metric application layers) It should also contain all data providers and rough description of plugins required in the data provider for push-forward data adapters. The integration overview should contain the EMDB buses required for this metric application as well as messages (names) that are transmitted over the buses in this metric application. The service overview should contain a rough description (goal) of each service. The section should also contain a description (maybe including a diagram) of the standard (dynamic) behavior of the metric application. The section should also provide a rough overview over all the external systems to this:

- ✓ What data providers are integrated.
- ✓ How are they integrated.
- ✓ What operation services are integrated (Monitoring, Logging, Directory Service, ...)

Furthermore, the section should contain the design decisions! (and alternatives!) Therefore it should at least provide information about:

- ✓ Discuss why certain adapter pattern are used.
- ✓ Discuss why certain metric kernel designs are used.
- ✓ Discuss possible extensions to the system at later stages.
- ✓ Discuss the rationals behind the decisions for specific services (the “service cut”)

B.3.2. Exception Behavior

It is very likely that in a real world scenario somethings may not always work as desired. Hence, it is important to discuss exceptions and errors in the design phase to increase

the robustness of each service and the whole metric application. This information is very closely related with quality problems and service status in the fine design section. Make sure to avoid duplications and prefer references!

This section in the design document should list all the exceptions including:

- ✓ Preconditions and environment for the exception.
- ✓ Steps that lead to the exception.
- ✓ Implications for the services and the whole metric application.
- ✓ Scenario identification for the users. That is: How does the metric customer, metric expert, or operator become aware of the exception.
- ✓ Definition of the logging behavior
- ✓ Definition of status changes for services (eg. out-of-sync status)
- ✓ Recovery steps (automatic recovery?)

B.3.3. Fine Design of the Integration

This section defines all the details for the integration of all the services required in this metric application. It should at least contain:

- ✓ Topics required for the integration (if more than the standard EMDB topics are required)
- ✓ Message hierarchy
- ✓ Message details including the fields and field properties (type, mandatory, and relations to other fields)

B.3.4. Fine Design of each Services

The design document should contain fine designs for all services to avoid coping with design problems and needing to make design decisions in the construction phase.

All services

- ✓ The static and dynamic architecture of the service.
- ✓ The design of the services should instantiate the reference architecture.

Service specific details

Additionally to the static and dynamic architecture the following data should be provided for the specific services

Visualization

- ✓ Data type / data structure required for the visualization.
- ✓ Sketch of visualization algorithm (eg. in pseudo code).
- ✓ Which visualization library to use.

Metric Kernel

- ✓ Iteratively design the static and dynamic architecture together.
- ✓ Design of the EUrEKA data API of the metric kernel (what data does the kernel need to provide to the visualization?, what parts of the calculation are variable).
- ✓ Design of message receivers: what messages does the kernel receive?
- ✓ Design of the controller and database: What data need to be stored, how to provide the metric values required, what transformations are required in the controller?

Data Adapter

- ✓ Type of the data adapter
- ✓ If push-forward: API design of the data adapter gateway.
- ✓ If invoke-push: Definition and details of the message that triggers the adapter.
- ✓ If pull-forward or invoke-push: Definition of the data access API in the data provider.

Data Quality

Metric Kernels and Data Adapters need to specify how data quality problems are handled and escalated.

- ✓ What data quality can be processed?
- ✓ Which quality problems can be identified by the metric kernel or data adapter?
- ✓ How (and what) quality problems are escalated over the EMDB)?

Data quality problems and exception behavior are very closely related. Typically every exception needs some investigation on potential data quality issues.

Monitoring and Service State

The service state is a rough means to communicate an overview of the current operation state to the metric operators. The service states need to be specified for each service, which typically includes a state automaton defining state transitions.

Furthermore, the performance indicators and their boundaries need to be defined for each service.

Logging

Define what important informations need to be visible in the log at which log level.

The section in the document should differentiate between technical and functional logging!

B.3.5. Tests

- ✓ Define all (actual) tests for this increment based on the design above.
- ✓ Group the tests by service and test stage.
- ✓ Use the test pattern and test templates developed in the bachelor thesis of Marco Moscher [Mos14] as well as the guides provided in section B.2.3.
- ✓ It is important to specify real tests by instantiating the test pattern rather then just to reference them.

C

Student Theses in the Context of this Thesis

The work on our metric system engineering approach is influenced by the hard work from numerous students who supported our goal with comprehensive research during their final thesis. A lot of additional details, evaluations, and discussions can be found in their theses. Thus, we heavily recommend reading their theses for further studies on these subjects.

C.1. Diploma Theses

Paul Tokarev (2012) Entwicklung einer Komponente zum Export von Analyseergebnissen aus MeDIC Dashboard

Driss El Majdoubi (2012) Entwicklung einer Erklärungskomponente für MeDIC Dashboard

Christian Charles (2012) Entwurf eines generischen Prozessleitstandes für Change Request Systeme

Martin Mädler (2012) Variabilität von Metriken und Dashboard-Items im Umfeld von MeDIC

Andrea Hutter (2012) Fachliche Integration von Metrik-Dashboards und Dashboard-Vorlagen in bestehende Software-Projekte

Andreas Steffens (2012) Entwurf eines Architekturmodells zur Integration heterogener Systeme in MeDIC

Steffen Conrad (2012) Ein Ansatz zum modellgetriebenen Test von EJB-basierten Informationssystemen

Tobias Löwenthal (2011) Generierung von web-basierten Prototypen für Geschäftsanwendungen

C.2. Master Theses

Elena Emelyanova (2014) Tool-Supported Project Prediction

Martin Lang (2014) Entwicklung einer Konstruktionsunterstützung für Messinfrastrukturen auf Basis des EMI-Frameworks

Thanh Vi Bach (2013) Entwurf prognostischer Softwareprozessmetriken auf Basis iterativen Clusterings

Endri Gjino (2013) Visually Assisted Mining for Smells in Change Request Systems

Thomas Röllig (2013) Konfigurationsunterstützung und Infrastrukturansatz für flexible Metrik-basierte Monitoring-Dashboards

Frederic Evers (2012) Konzeptionelle Erweiterung von Projektdashboards für unerfahrene Anwender

Meiliana (2011) Enhancing the MeDIC Meta-Models by EJB Conformant Variability Concepts

Ohm Samkoses (2011) Evaluating Presentation Layer Development Frameworks for EJB Applications in J2EE Architecture

Elena Soldatova (2011) Investigating the Impact of Refactoring and Reuse in Function Points

Stefan Cholakov (2011) Conception of Collaborative Project Cockpits with Integrated Interpretation Aids

Ricardo Tavizon (2011) Systematic Tool Supported Tailoring of Metrics

C.3. Bachelor Theses

Sebastian Rabenhorst (2014) Implementierung und Evaluierung von Performanceoptimierungen am Ticketanalyse Werkzeug RiVER

Bastian Greber (2014) Development of Multi-Project Metric Dashboards for Heterogeneous Tool Environments

Gordon Lawrenz (2014) Development of a Data Loss Prevention and Simulation Environment for RESTful Services

Marco Moscher (2014) Testpattern für EMI Metric Services am Beispiel von Wiki-Metriken

Jan Döring (2014) Entwicklung einer generischen Logging Infrastruktur für EMI-basierte Messsysteme

Arthur Otto (2013) Integration einer Metrik-Infrastruktur in die Projektverwaltung SSE-Lab

Ahmet Yüksektepe (2013) Entwicklung einer Service-Monitoring-Infrastrukur für MeDIC

Nick Russler (2013) Entwicklung einer Infrastruktur zur Fachlichen Integration von heterogenen Datenquellen in MeDIC

Matthias Gora (2012) Entwicklung eines Dashboard Prototyping-Werkzeugs

Elena Emelyanova (2012) Regelbasierte Initialisierung von Projektdashboards

Stefan Guha (2012) Entwicklung einer rollenspezifischen Benutzerschnittstelle zur Konfiguration von Metrik-Dashboards

Christian Hans (2012) Einsatz von Metrik-Dashboards im industriellen Umfeld

Michael Schlimnat (2012) Konzeption einer Internetplattform zur Dokumentation von Metriken

Michael Krein (2012) Generierung von Systemtests für Web-basierte Informationssysteme

Tristan Langer (2012) Erweiterung des Gargoyle Codegenerators um Semantische Beziehungen

Claude Mangen (2012) Generation of JSF-based Graphical User Interfaces for Web-based Information Systems

Christoph Greven (2011) Realisierung eines Werkzeugs zum Management von messpezifischen Projektaktivitäten

Frederic Evers (2010) Realisierung eines web-gestützten service-basierten Ideenmanagement-Systems für Metriken

Bibliography

[ABB00] F Arbab, Fs De Boer, and Mm Bonsangue. A logical interface description language for components. In *4th International Conference on Coordination Languages and Models, COORDINATION 2000 Limassol, Cyprus*, pages 249–266, Limassol, Cyprus, 2000. Springer Berlin Heidelberg.

[ABT00] Klaus-Dieter Althoff, Frank Bomarius, and Carsten Tautz. Knowledge Management for Building Learning Software Organizations. *Information Systems Frontiers*, 2(3-4):349–367, 2000.

[AKCK05] H Al-Kilidar, K Cox, and B Kitchenham. The use and usefulness of the ISO/IEC 9126 quality standard. In *Empirical Software Engineering, 2005. 2005 International Symposium on*, pages 7 pp.+, 2005.

[Anb04] F.T. Anbari. Earned value project management method and extensions. *IEEE Engineering Management Review*, 32(3):97–97, 2004.

[AS07] Scott W Ambler and Pramod J Sadalage. *Refactoring Databases, Evolutionary Database Design.* The Addison-Wesley Signature Series. Addison-Wesley, 4 edition, aug 2007.

[AW00] Rakesh Agrawal and Edward L. Wimmers. A Framework for Expressing and Combining Preferences. In *SIGMOD '00 Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 297–306, 2000.

[Bah09] A. Terry Bahill. What Is Systems Engineering?, 2009.

[Bas92] Victor R Basili. Software modeling and measurement: the Goal/Question/Metric paradigm, 1992.

[BBHM95] J. Bacon, J. Bates, R. Hayton, and K. Moody. Using events to build distributed applications. In *Second International Workshop on Services in Distributed and Networked Environments*, pages 148–155. IEEE Comput. Soc. Press, 1995.

[BDKZ93] Rajiv D. Banker, Srikant M. Datar, Chris F. Kemerer, and Dani Zweig. Software complexity and maintenance costs. *Communications of the ACM*, 36(11):81–94, nov 1993.

[BEM96] L Briand, K El Emam, and S Morasca. On the application of measurement theory in software engineering. *Empirical Software Engineering*, 1:61–88, 1996.

[Ber11] PA Bernstein. Generic schema matching, ten years later. *Techniques*, 4(11):695–701, 2011.

[Bey07] Mark A Beyer. Architecture Issues for Real-Time Data Warehousing. *Reproduction*, (July), 2007.

[BGR05] Rainer Berbner, Tobias Grollius, and Nicolas Repp. An approach for the Management of Service-oriented Architecture (SoA) based Application Systems. In *Enterprise Modelling and Information Systems Architectures (EMISA)*, pages 208–221, 2005.

[BR88] V.R. Basili and H.D. Rombach. The TAME project: towards improvement-oriented software environments. *IEEE Transactions on Software Engineering*, 14(6):758–773, jun 1988.

[BR91] V.R. Basili and H.D. Rombach. Support for comprehensive reuse. *Software Engineering Journal*, 6(5):303–316, jul 1991.

[BR00] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution. In *Proceedings of the conference on The future of Software engineering - ICSE '00*, pages 73–87, New York, New York, USA, may 2000. ACM Press.

[BR01] Philip a. Bernstein and Erhard Rahm. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, dec 2001.

[Bra14] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format, RFC 7159. Technical report, Internet Engineering Task Force (IETF), 2014.

[CCP07] Cristina Cachero, Coral Calero, and Geert Poels. Metamodeling the Quality of the Web Development Process Intermediate Artifacts. In *Web Engineering*, pages 74–89, Berlin, Heidelberg, 2007. Springer-Verlag.

[CD97] Surajit Chaudhuri and U Dayal. An overview of data warehousing and OLAP technology. *ACM Sigmod record*, (March 1997), 1997.

[CET07] Manuel Clavel, Marina Egea, and Viviane Torres. Model Metrication in MOVA: A Metamodel-Based Approach using OCL. Technical report, Information Security Group, ETH Zürich, 2007.

[Cha04] D A Chappell. *Enterprise Service Bus: Theory in Practice*. Theory in practice. O'Reilly Media, 2004.

[Cha12] Christian Charles. Entwurf eines generischen Prozessleitstandes für Change Request Systeme, 2012.

[CHM+07] M. Ciolkowski, J. Heidrich, J. Munch, F. Simon, and M. Radicke. Evaluating Software Project Control Centers in Industrial Environments. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pages 314–323. Ieee, 2007.

[CHM08] Marcus Ciolkowski, Jens Heidrich, and Jürgen Münch. Practical Guidelines for Introducing Software Cockpits in Industry. In *4th Software Measurement European Forum (SMEF 2008)*, may 2008.

[CHNP90] Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feasibility Study Feature-Oriented Domain Analysis (FODA) Kyo C . Kang. *Distribution*, (November), 1990.

[Cho11] Stefan Cholakov. *Conception of Collaborative Project Cockpits with Integrated Interpretation Aids*. Master, RWTH Aachen University, 2011.

[CHSR08] M. Ciolkowski, J. Heidrich, F. Simon, and M. Radicke. Empirical results from using custom-made software project control centers in industrial environments. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 243–252. ACM, 2008.

[CK94] S R Chidamber and C F Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[CMRW07] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core language. *W3C recommendation*, 26(May):1–85, 2007.

[Con12] Steffen Conrad. *Ein Ansatz zum modellgetriebenen Testen von EJB-basierten Informationssystemen*. PhD thesis, 2012.

[CSP13] Anton Chuvakin, Kevin Schmidt, and Chris Phillips. *Logging and Log Management: The Authoritative Guide to Understanding the Concepts Surrounding Logging and Log Management*. Syngress Publishing, 2013.

[CSS09] Irina Diana Coman, Alberto Sillitti, and Giancarlo Succi. A Case-study on Using an Automated In-process Software Engineering Measurement and Analysis System in an Industrial Environment. In *31st International Conference on Software Engineering (ICSE 2009)*, pages 89–99, Vancouver, BC, 2009. IEEE Computer Society.

[DBK+06] Reiner R Dumke, René Braungarten, Martin Kunz, Andreas Schmietendorf, and Cornelius Wille. Strategies and Appropriateness of Software Measurement Frameworks. In *Proceedings of the International Conference on Software Process and Product Measurement (MENSURA 2006)*, pages 150–170, 2006.

[DdOdlP98] Juan C. Dueñas, William L. de Oliveira, and Juan A. de la Puente. A Software Architecture Evaluation Model. In *Proceedings of the Second International ESPRIT ARES Workshop Las Palmas de Gran Canaria, Spain February 26–27*, pages 148–157. Springer Berlin Heidelberg, 1998.

[Dek99] Carol a. Dekkers. The Secrets of Highly Successful Measurement Programs. *Cutter IT Journal*, 12(4):29–35, 1999.

[DK01] Reiner Dumke and Reinhard Koeppe. Conception of a Web-Based SPE Development Infrastructure. In *Performance Engineering, State of the Art and Current Trends*, pages 1–19. Springer, 2001.

[DKW00] Reiner R Dumke, Reinhard Koeppe, and Cornelius Wille. *Software Agent Measurement and Self-Measuring Agent-Based Systems*. 2000.

[DLGP08a] M. Diaz-Ley, F. Garcia, and M. Piattini. Implementing Software Measurement Programs in Non Mature Small Settings. *Software Process and Product Measurement*, pages 154–167, 2008.

[DLGP08b] María Diaz-Ley, Félix García, and Mario Piattini. MIS-PyME Software Measurement Maturity Model: Supporting the Definition of Software Measurement Programs. In A. Jedlitschka and O. Salo, editors, *PROFES '08: Proceedings of the 9th international conference on Product-Focused Software Process Improvement*, pages 19–33, Berlin, Heidelberg, 2008. Springer-Verlag.

[DLGP10] María Díaz-Ley, Félix García, and Mario Piattini. MIS-PyME software measurement capability maturity model - Supporting the definition of software measurement programs and capability determination. *Advances in Engineering Software*, 41(10-11):1223–1237, oct 2010.

[DN06] Valentin Dinu and Prakash Nadkarni. Guidelines for the effective use of entity-attribute-value modeling for biomedical databases. *International journal of medical informatics*, 76(11-12):769–79, 2006.

[dO03] M. de Los Angeles Martin and Luis Olsina. Towards an Ontology for Software Metrics and Indicators as the Foundation for a Cataloging Web System. In *Web Congress*, pages 103–113. IEEE Computer Society Press, 2003.

[Dör14] Jan Simon Döring. *Development of a generic Logging Infrastructure for EMI-based measurement systems*. Bachelor thesis, RWTH Aachen University, 2014.

[DSZ06] R Dumke, Andreas Schmietendorf, and Horst Zuse. Formal Description of Software Measurement and Evaluation - A Short Overview and Evaluation. Technical report, 2006.

[Dum12] Reiner R Dumke. Software-Messung und -Bewertung - Eine Bilanz, 2012.

[DYAG09] Reiner Dumke, Hashem Yazbek, Evan Asfoura, and Konstantina Georgieva. A General Model for Measurement Improvement. In *Lecture notes in computer science, IWSM/Mensura 2009*, number 2, pages 48–61, 2009.

[Dyb05] T. Dyba. An empirical investigation of the key factors for success in software process improvement. *IEEE Transactions on Software Engineering*, 31(5):410–424, may 2005.

[Eck95a] Wayne Eckerson. Three Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client Server Applications. *Open Information Systems*, 10(1), 1995.

[Eck95b] Wayne W. Eckerson. Three Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client Server Applications. *Open Information Systems*, 3(20):46–50, 1995.

[Eck06] Wayne W Eckerson. *Performance Dashboards: Measuring, Monitoring, and Managing Your Business*. John Wiley & Sons, 2006.

[EDBS04] Christof Ebert, Reiner Dumke, Manfred Bundschuh, and Andreas Schmietendorf. *Best Practices in Software Measurement*. 2004.

[EdNBK01] Clark Evans, Ingy döt Net, and Oren Ben-Kiki. xmlhack: Evans moves against angle brackets in MinML, 2001.

[Ele90] Inc Electronics Engingeers. IEEE Standard Glossary of Software Engineering Terminology. *Office*, 121990:84, 1990.

[Eve10] Frederic Evers. *Realization of a web supported service based idea-management-system for metrics*. Bachelor thesis, RWTH Aachen University, 2010.

[Fen94] N. Fenton. Software measurement: a necessary scientific basis. *IEEE Transactions on Software Engineering*, 20(3):199–206, mar 1994.

[Few06] Stephen Few. *Information Dashboard Design: The Effective Visual Communication of Data*. O'Reilly Media, Inc., 2006.

[Few12] Stephen Few. *Show Me the Numbers*. 2012.

[Fie00] RT Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.

[Fit] John Fitch. Turn your business dashboard into a cockpit.

[FL14] Martin Fowler and James Lewis. Microservices, 2014.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, nov 2002.

[FP98] Norman E. Fenton and Shari L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 1998.

[FT02] Roy T. Fielding and Richard N. Taylor. Principled design of the modern Web architecture. *ACM Transactions on Internet Technology*, 2(2):115–150, may 2002.

[GDPG03] M.P. Papazoglou Georgakopoulos, D., M P Papazoglou, and D Georgakopoulos. Service-Oriented Computing. *Communications of the ACM*, 46(10):24–28, 2003.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, 1995.

[Gji13] Endri Gjino. *Visually Assisted Mining for Smells in Change Request Systems*. Master thesis, RWTH Aachen University, 2013.

[Gor13] Matthias Gora. *Bachelorarbeit Entwicklung eines Dashboard Prototyping-Werkzeugs Construction of a dashboard prototyping-tool*. Bachelor thesis, RWTH Aachen University, 2013.

[Gre11] Christoph Greven. *Realisierung eines Werkzeugs zum Management von messpezifischen Projektaktivitäten*. Bachelor thesis, RWTH Aachen University, 2011.

[GSC+07] F Garcia, M Serrano, J Cruzlemus, F Ruiz, and M Piattini. Managing software process measurement: A metamodel-based approach. *Information Sciences*, 177(12):2570–2586, 2007.

[HAB05] AY Halevy, Naveen Ashish, and Dina Bitton. Enterprise information integration: successes, challenges and controversies. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 778–787. ACM, 2005.

[Han12] Christian Hans. *Einsatz von Metrik-Dashboards im industriellen Umfeld*. Bachelor thesis, RWTH Aachen University, 2012.

[Hei08] Jens Heidrich. *PhD Theses in Experimental Software Engineering*. Phd thesis, Technische Universität Kaiserslautern, 2008.

[HF97] T. Hall and N. Fenton. Implementing effective software metrics programs. *IEEE Software*, 14(2):55–65, 1997.

[HKR12] Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A plug-in-based framework for web-based project portals. In *TOPI 2012*, pages 61–66. IEEE, 2012.

[HL11] Jairus Hihn and Scott Lewicki. Bootstrapping Process Improvement Metrics: CMMI Level 4 Process Improvement Metrics in a Level 3 World. In *Proceedings of the 44th Hawaii International Conference on System Sciences*, pages 1–10, 2011.

[HM04] Jens Heidrich and Jürgen Münch. Software project control centers: concepts and approaches. *Journal of Systems and Software*, 70(1-2):3–19, 2004.

[HM07] Jens Heidrich and Jürgen Münch. Cost-Efficient Customisation of Software Cockpits by Reusing Configurable Control Components. In *4th Software Measurement European Forum (SMEF 2007)*, may 2007.

[HM08a] J. Heidrich and J. Münch. Goal-oriented setup and usage of custom-tailored software cockpits. In *PROFES '08: Proceedings of the 9th international conference on Product-Focused Software Process Improvement*, pages 4–18. Springer-Verlag, 2008.

[HM08b] Jens Heidrich and Jürgen Münch. Goal-oriented setup and usage of custom-tailored software cockpits. In *PROFES '08: Proceedings of the 9th international conference on Product-Focused Software Process Improvement*, pages 4–18. Springer-Verlag, 2008.

[HMO08] Lasse Harjumaa, Jouni Markkula, and Markku Oivo. How Does a Measurement Programme Evolve in Software Organizations? In *PROFES '08: Proceedings of the 9th international conference on Product-Focused Software Process Improvement*, pages 230–243, Berlin, Heidelberg, 2008. Springer-Verlag.

[HMW06a] Jens Heidrich, J. Münch, and Axel Wickenkamp. Usage Scenarios for Measurement-based Project Control. In *Proceedings of the 3rd Software Measurement European Forum (SMEF 2006),(Ton Dekkers, Ed.)*, pages 47–60, 2006.

[HMW06b] Jens Heidrich, J. Münch, and Axel Wickenkamp. Usage Scenarios for Measurement-based Project Control. In *Proceedings of the 3rd Software Measurement European Forum (SMEF 2006),(Ton Dekkers, Ed.)*, pages 47–60, 2006.

[Hor14] Thorsten Horn. EAI Enterprise Application Integration. \url{http://www.torsten-horn.de/techdocs/eai.htm#Integrationstopologien}, nov 2014.

[HT99] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional, 1st editio edition, 1999.

[Hug13] James Hughes. Micro Service Architecture, 2013.

[Hüt12] Michael Hüttermann. *DevOps for Developers*. Apress, Berkeley, CA, 2012.

[Hut13] Andrea Hutter. *Business Integration of Metric-Dashboards and Dashboard-Templates for existing Software-Projects*. Diploma thesis, RWTH Aachen University, 2013.

[HW03a] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[HW03b] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Int13] ECMA International. ECMA-404: The JSON Data Interchange Format. Technical report, Geneva, Switzerland, 2013.

[ISO03] ISO/IEC 9126:2003. Product Quality Series, 2003.

[ISO05] ISO/IEC. ISO/IEC 25000 - Software engineering - Software product Quality Requirements and Evaluation (SQuaRE) - Guide to SQuaRE. Technical report, 2005.

[ISO07] ISO/IEC 15939:2007. Software Engineering - Software Measurement Process, 2007.

[ISO08] ISO/IEC 12207:2008. Standard for Systems and Software Engineering - Software Life Cycle Processes, 2008.

[ISO14] ISO/IEC 25000:2014. Systems and software Quality Requirements and Evaluation, 2014.

[Joh01] Philip Johnson. You can't even ask them to push a button: Toward ubiquitous, developer-centric, empirical software engineering. In *Visions for Software Design and Productivity: Research and Applications*, 2001.

[Jon06] Capers Jones. The Economics of Software Maintenance in the Twenty First Century, 2006. *Performance Engineering to Enhance the Maintenance*, 2006.

[Jos07] Nicolai M. Josuttis. *SOA in Practice: The Art of Distributed System Design*. "O'Reilly Media, Inc.", 2007.

[Kai02] Michael Kaib. *Enterprise Application Integration*, volume 45. Deutscher Universitäts-Verlag, 2002.

[Kir01] Colin Kirsopp. Measurement and the software development process. In *12th European Software Control and Metrics Conference*, page 165. Citeseer, 2001.

[KMZB08] Martin Kunz, Steffen Mencke, Niko Zenker, and René Braungarten. Quality-Driven Orchestration of Services. In Reiner R. Dumke, René Braungarten, Günter Büren, Alain Abran, and Juan J. Cuadrado-Gallego, editors, *Software Process and Product Measurement, Proceedings of the*

International Conferences on Software Metrics IWSM 2008, Metrikon 2008, and Mensura 2008, pages 26–35, München, 2008. Springer.

[KN92] R S Kaplan and D P Norton. The balanced scorecard–measures that drive performance. *Harvard Business Review*, 70(1):71–9, 1992.

[Kre12] Michael Krein. *Generierung von Systemtests für Web-basierte Informationssysteme.* PhD thesis, 2012.

[Kru95] P.B. Kruchten. The 4+1 View Model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[Kru02] C. Krueger. Variation management for software production lines. *Software Product Lines*, pages 107–108, 2002.

[KS06] Karen Kent and Murugiah P Souppaya. SP 800-92. Guide to Computer Security Log Management. Technical report, Gaithersburg, MD, United States, 2006.

[KSDW06] Martin Kunz, Andreas Schmietendorf, R Dumke, and Cornelius Wille. Towards a service-oriented measurement infrastructure. In *Proceedings of the 3rd Software Measurement European Forum (SMEF)*, pages 197–207, 2006.

[Kun09] Martin Kunz. *Framework for a Service-oriented Measurement Infrastructure.* Dissertation, Otto-von-Guericke-Universität Magdeburg, 2009.

[KWS+11] J.L. Krein, Patrick Wagstrom, S.M. Sutton Jr, Clay Williams, and C.D. Knutson. The problem of private information in large software organizations. In *Proceeding of the 2nd workshop on Software engineering for sensor network applications*, pages 218–222. ACM, 2011.

[KZR06] Rajiv Kishore, Hong Zhang, and R. Ramesh. Enterprise integration using the agent paradigm: foundations of multi-agent-based integrative business information systems. *Decision Support Systems*, 42(1):48–78, oct 2006.

[Lan12] Tristan Langer. *Erweiterung des Gargoyle Codegenerators um Semantische Beziehungen.* Bachelor thesis, RWTH Aachen University, 2012.

[Lan14] Martin Lang. *Entwicklung einer Konstruktionsunterstützung für Messinfrastrukturen auf Basis des EMI-Frameworks.* Master thesis, RWTH Aachen University, 2014.

[Lav00] L. Lavazza. Providing automated support for the GQM measurement process. *IEEE Software*, 17(3):56–62, 2000.

[Lav05] L. Lavazza. Automated support for process-aware definition and execution of measurement plans. *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 234–243, 2005.

[Law14] Gordon Lawrenz. *Development of a Data Loss Prevention and Simulation Environment for RESTful Services.* Bachelor thesis, RWTH Aachen University, 2014.

[LdBG08] Luigi A Lavazza, Vieri del Bianco, and Carla Garavaglia. Model-based functional size measurement. *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement - ESEM '08*, page 100, 2008.

[LDBK05] Mathias Lother, Reiner Dumke, René Braungarten, and Martin Kunz. Ein Portal zur Funktionalen Größenmessung von Software Anfänge eines Software eMeasurement. *Softwaretechnik Trends*, 25(1), 2005.

[LHM+09] Peter Liggesmeyer, Jens Heidrich, Jürgen Münch, Robert Kalcklösch, Henning Barthel, and Dirk Zeckzer. Visualization of Software and Systems as Support Mechanism for Integrated Software Project Control. *Human-Computer Interaction. New Trends, 13th International Conference, HCI International 2009, San Diego, CA, USA, July 19-24, 2009, Proceedings, Part I*, 5610:846–855, jan 2009.

[Lik04] Jeffrey K Liker. *The Toyota way: 14 management principles from the world's greatest manufacturer.* McGraw-Hill New York, 2004.

[Lim] The International Software Benchmarking Standards Group Limited. ISBSG Web Site.

[LL13] Jochen Ludewig and Horst Lichter. *Software engineering: Grundlagen, Menschen, Prozesse, Techniken.* dpunkt.verlag, Heidelberg, 3., korrigierte auflage edition, 2013.

[Löw11] Tobias Löwenthal. *Generierung von web-basierten Prototypen für Geschäftsanwendungen.* Diploma thesis, RWTH Aachen University, 2011.

[LWHS01] C. Lokan, T. Wright, P. Hill, and M. Stringer. Organizational benchmarking using the ISBSG Data Repository. *IEEE Software*, 18(5):26–32, 2001.

[Mäd12] Martin Mädler. *Variabilität von Metriken und Dashboard-Items im Umfeld von MeDIC.* Diploma thesis, RWTH Aachen University, 2012.

[Man12] Claude Mangen. *Generation of JSF-based Graphical User Interfaces for Web-based Information Systems.* Bachelorarbeit, RWTH Aachen, 2012.

[MB97] Sandro Morasca and LC Briand. Towards a theoretical framework for measuring software attributes. In *Software Metrics Symposium*, pages 119–126. IEEE Computer Society, 1997.

[MB00] M.G. Mendonsa and V.R. Basili. Validation of an approach for improving existing measurement frameworks. *IEEE Transactions on Software Engineering*, 26(6):484–499, jun 2000.

[Mei11] Meiliana. *Enhancing the MeDIC Meta-Models by EJB Conformant Variability Concepts*. Master thesis, RWTH Aachen University, 2011.

[Mel04] S Melnyk. Metrics and performance measurement in operations management: dealing with the metrics maze. *Journal of Operations Management*, 22(3):209–218, jun 2004.

[Met09] Sandi Metz. SOLID Object-Oriented Design, 2009.

[MGRP09] Beatriz Mora, Felix Garcia, Francisco Ruiz, and Mario Piattini. Model-Driven Software Measurement Framework: A Case Study. In *2009 Ninth International Conference on Quality Software*, pages 239–248. Ieee, aug 2009.

[Mic12] Sun Microsystems. Java Programming Language (1.5), 2012.

[MJCH08] M. Monperrus, J.M. Jézéquel, J. Champeau, and B. Hoeltzener. A model-driven measurement approach. In *Proceedings of the ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems (MODELS 2008)*, pages 505–519. Springer, 2008.

[MO07] Hernan Molina and Luis Olsina. Towards the Support of Contextual Information to a Measurement and Evaluation Framework. In *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*, pages 154–166. Ieee, sep 2007.

[Mos14] Marco Moscher. *Testpattern für EMI Metric Services am Beispiel von Wiki-Metriken*. Bachelor thesis, RWTH Aachen University, 2014.

[MP06] Jacquelin A McQuillan and James F Power. Towards re-usable metric definitions at the meta-level. In *PhD Workshop of the 20th European Conference on Object-Oriented Programming (ECOOP 2006)*, 2006.

[MSW] J. Mangler, E. Schikuta, and C. Witzany. Quo vadis interface definition languages? towards a interface definition language for restful services. In *2009 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 1–4.

[Mul13] Rita Mulcahy. *PMP Exam Prep: Rita's Course in a Book for Passing the PMP Exam*. McGraw-Hill Professional, 8th edition, 2013.

[Net10] Microsoft Developer Network. Removal of UDDI Services from Server Operating System (Windows), 2010.

[Net14] Microsoft Developer Network. Using a Three-Tier Architecture Model (COM+), 2014.

[NV01] Frank Niessink and Hans Van Vliet. Measurement program success factors revisited. *Information and Software Technology*, pages 1–25, 2001.

[ODR84] Ce. O'Meley, Sw. Draper, and Ms. Riley. Constructive Interaction: A Method for Studying Human-Computer-Human Interaction. In *Proceedings of IFIP Interact*, pages 269–274, 1984.

[OLP02] Luis Olsina, Guillermo Lafuente, and Oscar Pastor. Towards a reusable repository for web metrics. *Quality*, 1(1):061–073, 2002.

[OM04] Luis Olsina and María De Los Angeles Martín. Ontology for Software Metrics and Indicators: Building Process and Decisions Taken. In *Web Engineering*, page 778, 2004.

[OMF+03] L Olsina, M A Martin, J Fons, S Abrahao, and O Pastor. Towards the Design of a Metrics Cataloging System by Exploiting Conceptual and Semantic Web Approaches. In *Web Engineering*, pages 324–333, Heidelberg, 2003. Springer-Verlag.

[Ott13] Arthur Otto. *Integration einer Metrik-Infrastruktur in die Projektverwaltung SSE-Lab*. PhD thesis, 2013.

[PAFM04] Edgardo Palza, Alain Abran, Christopher Fuhrman, and Eduardo Miranda. V & V Measurement Management Tool for Safety-Critical Software. In *Proceedings of IWSM/MetriKon 2004*, 2004.

[Pau06] F. Paulisch. Establishing a Common Measurement System. In *Proceedings of the International Workshop on Software Metrics and DASMA Software Metrik Kongress*, pages 1–3, 2006.

[PCN+08] S.T. Parkinson, S. Counsell, M. Norman, R. M. Hierons, and M. Lycett. The precursor to an industrial software metrics program. In *ITI 2008 - 30th International Conference on Information Technology Interfaces*, pages 221–226. Ieee, jun 2008.

[Per00] Alan Perkins. Critical Success Factors for Enterprise Architecture Engineering. Technical report, 2000.

[Pfl93] S.L. Pfleeger. Lessons learned in building a corporate metrics program. *IEEE Software*, 10(3):67–74, may 1993.

[Pia07] Mario Piattini. Software Measurement Programs in SMEs - Defining Software Indicators: A Methodological Framework. In J. Münch and P. Abrahamsson, editors, *PROFES '07: Proceedings of the 8th international conference on Product-Focused Software Process Improvement*, pages 247–261, Berlin, Heidelberg, 2007. Springer-Verlag.

[PTDL08] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-Oriented Computing: a Research Roadmap. *International Journal of Cooperative Information Systems*, 17(02):223–255, jun 2008.

[Rab15] Sebastian Rabenhorst. *Implementation and Evaluation of Performance Optimizations in the Ticket Analysis Tool RiVER.* Bachelor thesis, RWTH Aachen University, 2015.

[Rem07] Ulrich Remus. Critical success factors for implementing enterprise portals: A comparison with ERP implementations. *Business Process Management Journal*, 13(4):538—-552, 2007.

[RMGW05] Roger Strukhoff, Lori MacVittie, Carmen Gonzalez, and Elizabeth White. Microsoft, IBM, SAP To Discontinue UDDI Web Services Registry Effort, 2005.

[RNC+10] S Russell, P Norvig, J.F. Candy, J.M. Malik, and D.D. Edwards. *Artificial Intelligence: A modern approach.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., third edit edition, 2010.

[Röl13] Thomas Röllig. *Konfigurationsunterstützung und Infrastrukturansatz für flexible Metrik-basierte Monitoring-Dashboards.* Master thesis, RWTH Aachen University, 2013.

[Rom91] H.D. Rombach. Practical benefits of goal-oriented measurement. *Software Reliability and Metrics*, pages 217–235, 1991.

[Rom05] Dieter Rombach. Integrated software process and product lines. In *International Software Process Workshop (SPW) 2005, Beijing*, pages 83–90. Springer, 2005.

[RSD07] D Rud, A Schmietendorf, and R Dumke. Resource metrics for service-oriented infrastructures. In *Proc. SEMSOA 2007*, pages 90–98, 2007.

[Rus13] Nick Russler. *Development of an Infrastructure for the Business Integration of Heterogeneous Data Sources in the EMI.* Bachelor thesis, RWTH Aachen University, 2013.

[Sch12] Bastian Schwartz. *Konzeption einer Komponenten-Kompo[1] B. Schwartz, "Konzeption einer Komponenten-Kompositionsinfrastruktur," RWTH Aachen University, 2012.sitionsinfrastruktur.* Diploma thesis, RWTH Aachen University, 2012.

[Sei03] Siegfried Seibert. Softwaremessung, quantitative Projektsteuerung und Benchmarking. *Projektmanagement, Jg*, 14:26–34, 2003.

[Sel05] Richard W Selby. Measurement-Driven Dashboards Enable Leading Indicators for Requirements and Design of Large-Scale Systems. In *IEEE METRICS*, page 22, 2005.

[SGN08] Ulrik Schroeder, Eva Giani, and Michael Nelles. Script und Folien der Vorlesung Web Engineering. 2008.

[SM07] Miroslaw Staron and Wilhelm Meding. A Modeling Language for Specifying and Visualizing Measurement Systems for Software Metrics. pages 300–307, 2007.

[SMN09] Miroslaw Staron, Wilhelm Meding, and Christer Nilsson. A framework for developing measurement systems and its industrial evaluation. *Inf. Softw. Technol.*, 51(4):721–737, 2009.

[Som11] Ian Sommerville. *Software Engineering*. 9th editio edition, 2011.

[Son14] Sonar Qube, 2014.

[ST] Klaus-Dieter Schewe and Bernhard Thalheim. Component-Driven Engineering of Database Applications.

[Sta03] Prof Stafford. Software Maintenance As Part of the Software Life Cycle. *Time*, 2003.

[Ste13] Andreas Steffens. *Entwurf eines Architekturmodells zur Integration heterogener Systeme in MeDIC*. Diploma thesis, RWTH Aachen University, 2013.

[Tav11] Ricardo Tavizon. *Systematic Tool Supported Tailoring of Metrics*. Master thesis, RWTH Aachen University, 2011.

[Tea10] CMMI Product Team. CMMI® for Development, Version 1.3 CMMI-DEV, V1.3. Technical Report November, 2010.

[The97] The National Performance Review (now the National Partnership for Reinventing Government). *Serving the American Public: Best Practices in Performance Measurement: Benchmarking Study Report*. Government Printing Office, Washington, D.C., 1997.

[Tho05] G Thomas. Mediation and enterprise service bus: A position paper. *on Mediation in*, pages 67–80, 2005.

[US08] Medha Umarji and Carolyn Seaman. Why Do Programmers Avoid Metrics? In *ESEM 08*, pages 129–138, Kaiserslautern, 2008. ACM.

[USE] Medha Umarji, Carolyn Seaman, and Henry H Emurian. Acceptance Issues in Metrics Program Implementation. *Information Systems*.

[VBR09] Matthias Vianden, Frank Berretz, and Tobias Rötschke. Werkzeugunterstützung für iterative Modernisierungsprozesse. In *WSR '09: Proceedings of the Workshop on Software Reegineering*, 2009.

[Via08] Matthias Vianden. *Entwurf und Realisierung eines Ansatzes zur Modernisierung der Architektur eines formularbasierten Informationssystems*. Diploma thesis, RWTH Aachen University, 2008.

[Vin97] S. Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 35(2):46–55, 1997.

[VL14] Matthias Vianden and Horst Lichter. Lessons Learned on Systematic Metric System Development at a large IT Service Provider. In *Proceeding of the 2nd International Workshop on Quantitative Approaches to Software Quality (QuASoC), in conjunction with 21th Asia-Pacific Software Engineering Conference (APSEC 2014)*, Jeju, South Korea, 2014. IEEE Computer Society.

[VLHH13] Matthias Vianden, Horst Lichter, Andrea Hutter, and Christian Hans. Engineering Metric-based Monitoring Dashboards - Development Process and Best Practices. In *Proceedings of the 20th Asia-Pacific Software Engineering Conference*, Bangkok, Thailand, 2013.

[VLJ13] Matthias Vianden, Horst Lichter, and Simona Jeners. History and Lessons Learnt from a Metrics Program at a CMMI Level 3 Company. In *Proceedings of 20th Asia-Pacific Software Engineering Conference, APSEC 2013, Vol. 2*, number CMMI, 2013.

[VLN12] Matthias Vianden, Horst Lichter, and Karl-Joachim Neumann. Towards Systematic Reuse of Metric Specifications. *International Journal of Digital Content Technology and its Applications*, 6(21):43–49, nov 2012.

[VLR09] Matthias Vianden, Horst Lichter, and Tobias Rötschke. Applying Test Case Metrics in a Tool Supported Iterative Architecture and Code Improvement Process. In *IWSM '09: Proceedings of the international Conference on Software Metrics*, Berlin, Heidelberg, 2009. Springer-Verlag.

[VLS13] Matthias Vianden, Horst Lichter, and Andreas Steffens. Towards a Maintainable Federalist Enterprise Measurement Infrastructure. In *Joint Conference of the 23nd International Workshop on Software Measurement (IWSM) and the 8th International Conference on Software Process and Product Measurement (Mensura)*, Ankara, Turkey, 2013.

[VLS14] Matthias Vianden, Horst Lichter, and Andreas Steffens. Experience on a Microservice-Based Reference Architecture for Measurement Systems. In *2014 21st Asia-Pacific Software Engineering Conference*, volume 1, pages 183–190. IEEE, dec 2014.

[WBD+10] Stefan Wagner, Manfred Broy, Florian Deißenböck, Michael Kläs, Peter Liggesmeyer, Jürgen Münch, and Jonathan Streit. Softwarequalitätsmodelle, 2010.

[Wes05] Linda Westfall. 12 Steps to Useful Software Metrics, 2005.

[WGH+15] Stefan Wagner, Andreas Goeb, Lars Heinemann, Michael Kläs, Constanza Lampasona, Klaus Lochmann, Alois Mayr, Reinhold Plösch, Andreas Seidl, Jonathan Streit, and Adam Trendowicz. Operationalised product quality models and assessment: The Quamoco approach. *Information and Software Technology*, 62(June 2015):101–123, 2015.

[WGT07] Thomas Wuttke, Peggy Gartner, and Steffi Triest. *Das PMP-Examen: fur die gezielte Prufungsvorbereitung*. Mitp-Verlag, Heidelberg, 2007.

[WLH+12] Stefan Wagner, Klaus Lochmann, Lars Heinemann, Michael Klas, Adam Trendowicz, Reinhold Plosch, Andreas Seidi, Andreas Goeb, and Jonathan Streit. The Quamoco product quality modelling and assessment approach. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1133–1142. IEEE, jun 2012.

[Woh12] C. Wohlin. Experimentation in Software Engineering. 2012.

[Woo01] Michael Wooldridge. Intelligent Agents: The Key Concepts. In *Multi-Agent Systems and Applications II*, volume 2322, pages 3–43. 2001.

[YDSN10] H. Yazbek, R. Dumke, A. Schmietendorf, and R. Neumann. Service-Oriented Measurement Infrastructure. In *2010 Eighth ACIS International Conference on Software Engineering Research, Management and Applications*, pages 303–308. IEEE, may 2010.

[Yük13] Ahmet Yüksektepe. *Entwicklung einer Service-Monitoring-Infrastruktur für die EMI*. Bachelor thesis, RWTH Aachen University, 2013.

[Zus91] Horst Zuse. *Complexity Measure: Measures and Methods*. de Gruyter, 1991.