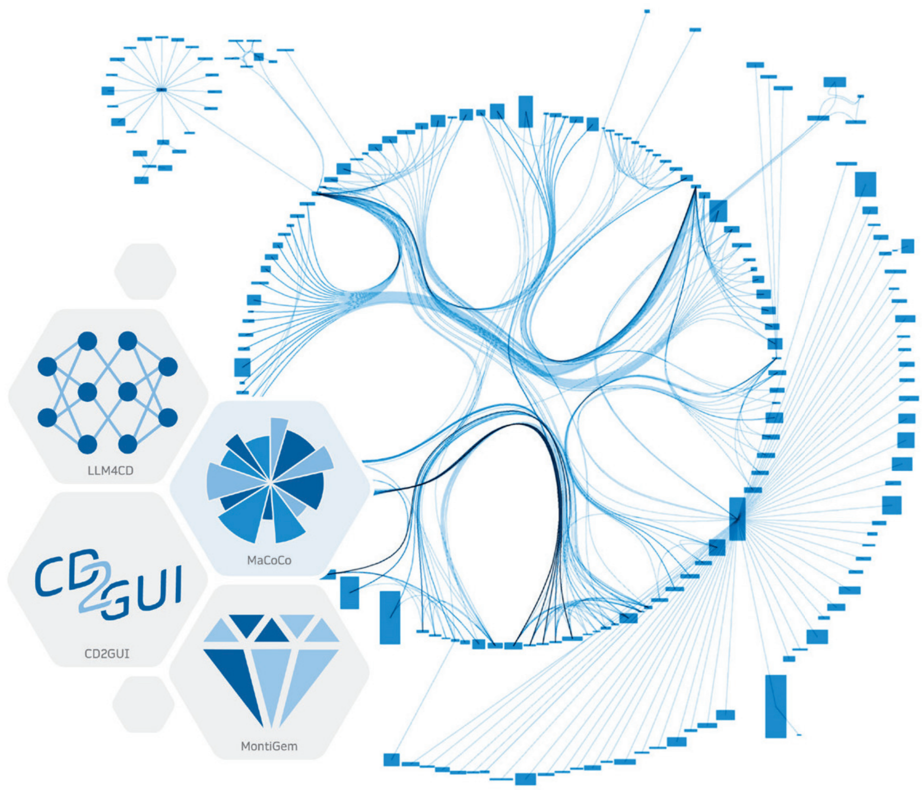
Lukas Netz

Model-Driven Method for the Development of Full-Size Web-Based Information Systems



Aachener Informatik-Berichte,
Software Engineering

Band 55

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

Model-Driven Method for the Development of Full-Size Web-Based Information Systems

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

vorgelegt von

Lukas Stephan Michael Netz,
M.Sc. RWTH
aus Düsseldorf

Berichter: Universitätsprofessor Dr. rer. nat. Bernhard Rumpe
Assistant Prof. Dr.rer.pol. Dominik Bork

Tag der mündlichen Prüfung: 7 Oktober 2024

Diese Dissertation ist auf den Internetseiten der Universitätsbibliothek online verfügbar.

[Net24] L. Netz:
Model-Driven Method for the Development of Full-Size Web-Based Information Systems.
Aachener Informatik-Berichte, Software Engineering, Band 55,
ISBN 978-3-8440-9833-4, Shaker Verlag, Dec. 2024.

Aachener Informatik-Berichte, Software Engineering

herausgegeben von
Prof. Dr. rer. nat. Bernhard Rumpe
Software Engineering
RWTH Aachen University

Band 55

Lukas Netz
RWTH Aachen University

Model-Driven Method for the Development of sFull-Size Web-Based Information Systems

Shaker Verlag
Düren 2024

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Zugl.: D 82 (Diss. RWTH Aachen University, 2024)

Print-ISBN 978-3-8440-9733-7
PDF-ISBN 978-3-8440-9833-4
ISSN 1869-9170
eISSN 2944-6910
https://doi.org/10.2370/9783844098334

Shaker Verlag GmbH • Am Langen Graben 15a • 52353 Düren
Telefon: 02421 / 99 0 11 - 0 • Telefax: 02421 / 99 0 11 - 9
Internet: www.shaker.de • E-Mail: info@shaker.de

Eidesstattliche Erklärung

Lukas Netz
erklärt hiermit, dass diese Dissertation und die darin dargelegten Inhalte die eigenen sind und selbstständig, als Ergebnis der eigenen originären Forschung, generiert wurden. Hiermit erkläre ich an Eides statt

1. Diese Arbeit wurde vollständig oder größtenteils in der Phase als Doktorand dieser Fakultät und Universität angefertigt;
2. Sofern irgendein Bestandteil dieser Dissertation zuvor für einen akademischen Abschluss oder eine andere Qualifikation an dieser oder einer anderen Institution verwendet wurde, wurde dies klar angezeigt;
3. Wenn immer andere eigene- oder Veröffentlichungen Dritter herangezogen wurden, wurden diese klar benannt;
4. Wenn aus anderen eigenen- oder Veröffentlichungen Dritter zitiert wurde, wurde stets die Quelle hierfür angegeben. Diese Dissertation ist vollständig meine eigene Arbeit, mit der Ausnahme solcher Zitate;
5. Alle wesentlichen Quellen von Unterstützung wurden benannt;
6. Wenn immer ein Teil dieser Dissertation auf der Zusammenarbeit mit anderen basiert, wurde von mir klar gekennzeichnet, was von anderen und was von mir selbst erarbeitet wurde;
7. Teile dieser Arbeit wurden zuvor veröffentlicht und zwar in:

    [**NMB24**] L. Netz, J. Michael, B. Rumpe: From Natural Language to Web Applications: Using Large Language Models for Model-Driven Software Engineering. In Modellierung 2024, Volume 348, pp 179-195, Mar 2024.

    [**BDM+24**] N. Baumann, J. Diaz, J. Michael, L. Netz, H. Nqiri, J. Reimer, B. Rumpe: Combining retrieval-augmented generation and few-shot learning for model synthesis of uncommon DSLs. In Modellierung Satellite Events 2024, Mar 2024.

    [**BGK+24**] C. Buschhaus, A. Gerasimov, J. C. Kirchhof, J. Michael, L. Netz, B. Rumpe, S. Stüber: Lessons Learned from Applying Model-Driven Engineering in 5 Domains: The Success Story of the MontiGem Generator Framework. In: Science of Computer Programming, Volume 232, pp. 103033, Jan. 2024.

[FHM+23] S. Fur, M. Heithoff, J. Michael, L. Netz, J. Pfeiffer, B. Rumpe, A. Wortmann: Sustainable Digital Twin Engineering for the Internet of Production. In: Digital Twin Driven Intelligent Systems and Emerging Metaverse, E. Karaarslan, Ö. Aydin, Ü. Cali, M. Challenger (Eds.), pp. 101-121, Springer Nature Singapore, Apr. 2023.

[MNN+22] J. Michael, I. Nachmann, L. Netz, B. Rumpe, S. Stüber: Generating Digital Twin Cockpits for Parameter Management in the Engineering of Wind Turbines. In: Modellierung 2022, pp. 33-48, Gesellschaft für Informatik, Jun. 2022.

[DHM+22] M. Dalibor, M. Heithoff, J. Michael, L. Netz, J. Pfeiffer, B. Rumpe, S. Varga, A. Wortmann: Generating Customized Low-Code Development Platforms for Digital Twins. In: Journal of Computer Languages (COLA), Volume 70, Art. 101117, Elsevier, Jun. 2022.

[DGM+21] I. Drave, A. Gerasimov, J. Michael, L. Netz, B. Rumpe, S. Varga: A Methodology for Retrofitting Generative Aspects in Existing Applications. In: Journal of Object Technology (JOT), A. Pierantonio (Eds.), Volume 20, pp. 1-24, AITO - Association Internationale pour les Technologies Objets, Nov. 2021.

[GMNR21] A. Gerasimov, J. Michael, L. Netz, B. Rumpe: Agile Generator-Based GUI Modeling for Information Systems. In: Modelling to Program (M2P), A. Dahanayake, O. Pastor, B. Thalheim (Eds.), pp. 113-126, Springer, Mar. 2021.

[GMN+20] A. Gerasimov, J. Michael, L. Netz, B. Rumpe, S. Varga: Continuous Transition from Model-Driven Prototype to Full-Size Real-World Enterprise Information Systems. In: 25th Americas Conference on Information Systems (AMCIS 2020), B. Anderson, J. Thatcher, R. Meservy (Eds.), pp. 1-10, AIS Electronic Library (AISeL), Association for Information Systems (AIS), Aug. 2020.

[AMN+20b] K. Adam, J. Michael, L. Netz, B. Rumpe, S. Varga: Model-Based Software Engineering at RWTH Aachen University. In: 40 Years EMISA: Digital Ecosystems of the Future: Methodology, Techniques and Applications (EMISA'19), Volume P-304, pp. 183-188, LNI, Gesellschaft für Informatik e.V., May 2020.

[AMN+20a] K. Adam, J. Michael, L. Netz, B. Rumpe, S. Varga: Enterprise Information Systems in Academia and Practice: Lessons learned from a MBSE Project. In: 40 Years EMISA: Digital Ecosystems of the Future: Methodology, Techniques and Applications (EMISA'19), Volume P-304, pp. 59-66, LNI, Gesellschaft für Informatik e.V., May 2020.

[GHK+20] A. Gerasimov, P. Heuser, H. Ketteniß, P. Letmathe, J. Michael, L. Netz, B. Rumpe, S. Varga: Generated Enterprise Information Systems: MDSE for Maintainable Co-Development of Frontend and Backend. In: Companion Proceedings of Modellierung 2020 Short, Workshop and Tools & Demo Papers, J. Michael, D. Bork (Eds.), pp. 22-30, CEUR Workshop Proceedings, Feb. 2020.

[MNRV19] J. Michael, L. Netz, B. Rumpe, S. Varga: Towards Privacy-Preserving IoT Systems Using Model Driven Engineering. In: Proceedings of MODELS 2019. Workshop MDE4IoT, N. Ferry, A. Cicchetti, F. Ciccozzi, A. Solberg, M. Wimmer, A. Wortmann (Eds.), pp. 595-614, CEUR Workshop Proceedings, Sep. 2019.

[ANV+18] K. Adam, L. Netz, S. Varga, J. Michael, B. Rumpe, P. Heuser, P. Letmathe: Model-Based Generation of Enterprise Information Systems. In: Enterprise Modeling and Information Systems Architectures (EMISA'18), M. Fellmann, K. Sandkuhl (Eds.), Volume 2097, pp. 75-79, CEUR Workshop Proceedings, CEUR-WS.org, May 2018.

Aachen, den 26.6.2024

Lukas Netz

Abstract

The implementation of information systems poses a variety of challenges for developers and requires a large number of experts from different application domains. This work presents a methodology in the area of generative model-driven development (MDD) for web-based information systems. The methodology consists of three basic building blocks: LLM4CD, CD2GUI and MontiGem. The first building block LLM4CD is based on generative artificial intelligence (LLMs) and transforms natural language requirements and domain descriptions into class diagrams in CD4A notation. The second building block CD2GUI uses these class diagrams and derives models for user interfaces in GUI-DSL notation. The last building block is MontiGem. It uses both types of models to generate a data-centric information system.

An important part of this work is the adaptability of the generated artifacts, this mainly concerns the possibility for modelers to adapt the models generated by the AI and CD2GUI, and to adapt and extend the code generated by MontiGem without having to edit the generated code or models directly. This makes iterative work with the methodology possible without discarding adapted code or adapted models.

It is precisely this adaptability that facilitates the transition from an underspecified modeled software prototype to a full-size model-driven information system that can be actively used in industry. In addition to the three building blocks, this thesis also presents the domain-specific modeling languages GUI-DSL and CD4A. The methodology is evaluated and discussed both in parts and as a whole. The practical applicability of the methodology is demonstrated using the case study MaCoCo, an information system used by RWTH Aachen University, which serves as a robust benchmark for the validation of the methodology. In addition, other smaller projects are presented that offer a wide range of applications and valuable insights.

Kurzfassung

Die Implementierung von Informationssystemen stellt eine Vielzahl von Herausforderungen an die Entwickler und erfordert eine Vielzahl von Experten aus unterschiedlichen Anwendungsdomänen. In dieser Arbeit wird eine Methodik im Bereich der generativen modellgetriebenen Entwicklung (MDD) für webbasierte Informationssysteme vorgestellt. Die Methodik besteht aus drei grundlegenden Bausteinen: LLM4CD, CD2GUI und MontiGem. Der erste Baustein LLM4CD basiert auf generativer künstlicher Intelligenz (LLMs) und transformiert natürlichsprachige Anforderungen und Domänenbeschreibungen in Klassendiagramme in CD4A-Notation. Der zweite Baustein CD2GUI verwendet diese Modelle und leitet daraus Modelle für Benutzeroberflächen in GUI-DSL-Notation ab. Der letzte Baustein MontiGem verwendet beide Modellarten, um daraus ein datenzentrisches Informationssystem zu generieren.

Ein wichtiger Bestandteil dieser Arbeit ist die Anpassbarkeit der erzeugten Artefakte. Dies betrifft vor allem die Möglichkeit für Modellierer, sowohl die durch die KI und CD2GUI erzeugten Modelle, als auch den durch MontiGem erzeugten Code zu editieren und zu erweitern. Somit ist einen iteratives Arbeiten mit der Methodik möglich, ohne dass händisch angepasste Modelle oder Code bei jeder Iteration verworfen werden.

Genau diese Anpassbarkeit begünstigt den Übergang von einem unterspezifiziert modelliertem Software-Prototypen zu einem funktional vollständigen modellgetriebenen Informationssystem welches aktiv in Industrie verwendet werden kann.

In dieser Arbeit werden neben den drei Bausteinen auch die domänenspezifischen Modellierungssprachen GUI-DSL und CD4A erläutert. Die Methodik wird sowohl in Teilen als auch als Gesamtes evaluiert und diskutiert. Die praktische Anwendbarkeit der Methodik wird anhand der Fallstudie MaCoCo demonstriert. MaCoCo ist ein Informationssystem, das von der RWTH Aachen eingesetzt wird und als robuster Maßstab für die Validierung der Methodik dient. Darüber hinaus werden weitere kleinere Projekte vorgestellt, die ein breites Spektrum an Anwendungen und Erkenntnissen bieten.

Danksagung

Die Erstellung einer Dissertation ist ein langwieriges und herausforderndes Unterfangen. Es ist mir daher ein großes Anliegen, all jenen zu danken, die mich auf diesem Weg begleitet und unterstützt haben.

Ich möchte meinem Doktorvater, Prof. Dr. Bernhard Rumpe danken, der mir die Möglichkeit gab, an seinem Lehrstuhl zu forschen und zu promovieren. Die ausgiebigen Diskussionen mit ihm, waren entscheidend für den Fortschritt meiner Arbeit und haben mir geholfen, die wissenschaftlichen Inhalte von neuen Seiten zu betrachten. Ebenso möchte ich auch Professor Prof. Dr. Dominik Bork für seine Zeit und sein Feedback danken. Ein besonderer Dank gilt Prof. Dr. Erica Ábráham, die mir als studentische und wissenschaftliche Hilfskraft an Ihrem Lehrstuhl die Chance gab, an Forschungsprojekten und wissenschaftlichen Publikationen mitzuwirken. Ihre Leidenschaft für die Forschung war ansteckend und hat mich maßgeblich dazu inspiriert, die Promotion als spannende Option zu betrachten. Ein ganz besonderer Dank gebührt Dr. Judith Michael, meiner Gruppenleiterin am Lehrstuhl, die mich auf meinem gesamten Promotionsweg unterstützt hat. Ihre stets offene Tür und Bereitschaft auch nach Feierabend für Diskussionen und Fragen zur Verfügung zu stehen, war für mich von unschätzbarem Wert. Ebenfalls danken möchte ich meinen Kollegen, insbesondere Imke Nachmann, Arkadii Gerasimov, Joel Charles, Alex Hellwig, Malte Heithoff, Constantin Buschhaus, Marita Breuer, Marco Konersmann, Nico Jansen und Haron Nqiri, die nicht nur bereit waren, wertvolles Feedback zu dieser Arbeit zu geben sondern auch stets Zeit hatten, sich mit mir über die Forschungsthehmen auszutauschen.

Meinen Brüdern möchte ich ebenfalls meinen Dank aussprechen. Ihre ständige Unterstützung und das Wissen, dass ich mich jederzeit auf sie verlassen kann, waren mir eine große Stütze in allen Lebenslagen. Abschließend gilt mein größter Dank meinen Eltern, ohne deren Unterstützung und Ermutigung dieser akademische Weg für mich nicht möglich gewesen wäre.

Meinem Vater Priv. Doz. Dr. med. Dipl.-Phys. Johannes Netz.
In Gedenken an seine Zeit als Wissenschaftler und Arzt.

Aachen, April 2024
Lukas Netz

Contents

Chapter 1

Introduction

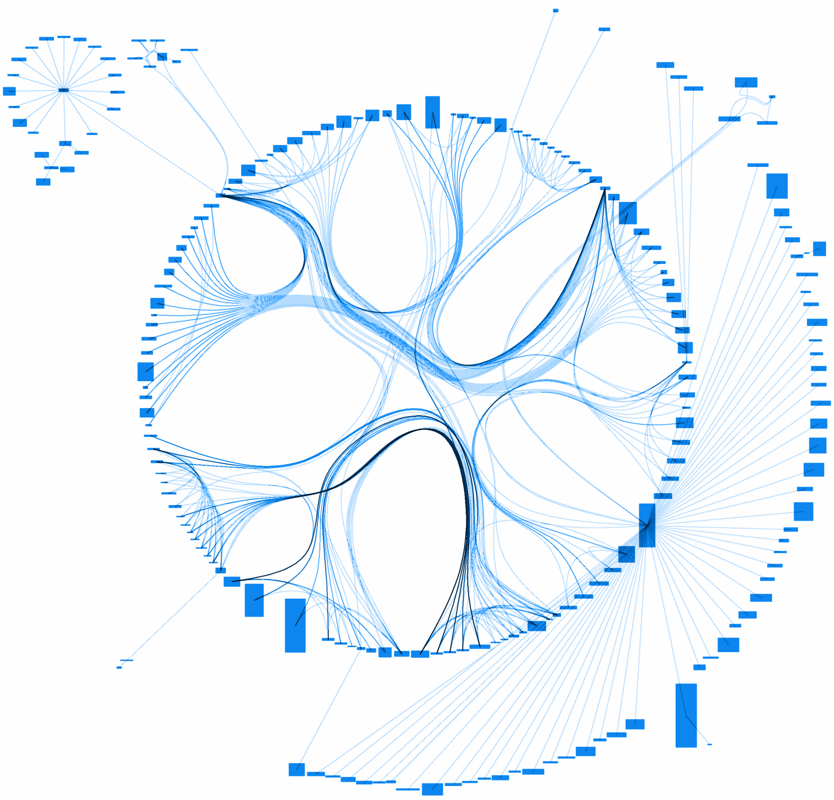


Figure 1.1: Cover Image: Zoomed out view on a class diagram of all persisted classes of the MaCoCo information system. The project will be presented in detail in Chapter 8

The widespread adoption of digital technology has fundamentally changed how we communicate, work, and handle information [BP20, Sch20]. It has reshaped software engineering, driving the need for complex, flexible systems focused on user experience [TWT22, XX13]. This shift, widely known as digitalization [BK16], is a defining feature of both our personal lives and professional environments. The COVID-19 pandemic has significantly accelerated the global shift toward digitalization, leading to an increased demand for Information Systems (IS) [DAKM20, JHC21, ST19]. These systems are crucial for integrating various processes and data, enabling businesses to operate more efficiently and innovatively. Developing such sophisticated software systems requires a blend of specialized software development skills and in-depth understanding of the specific domain [FR07]. A zoomed-out view of the underlying data structure of such a complex system (MaCoCo) developed during this thesis is shown in Figure 1.1. This creates a challenge for organizations as they strive to quickly implement these systems while also managing the investment of time and resources involved. The modern industry is a fast-paced environment that necessitates rapid development and presentation of products. The ability to prototype rapidly and validate ideas before full-scale development can save precious time and resources [HWDB20], offering businesses the agility they need to stay ahead. There are platforms designed to streamline the creation of IS, such as Low-Code Development platforms (LCDP) [PRH21, DRKdL+22], which aim to simplify the software development process. However, the issue with the available platforms is that they still demand a considerable level of software engineering expertise to effectively utilize them [HSVT21]. Our recent research in software engineering has led to new tools [MNN+22, DHM+22], languages [GMNR21], and methods [DGM+21] that provide more efficient methods to build complex IS [GMN+20, BGK+23a]. Our exploration and application of Large Language Models (LLMs) [CGR23], points out the potential for individuals lacking formal software skills, to define these systems by only using natural language [MFBF+22].

1.1 Motivation and Problem Statement

There is a clear need for a solid approach and a set of tools to build ISs that reduce the workload of professional software developers. As digital technology becomes a standard part of business, we need systems that can be at least partially developed by those without training in software engineering or programming. Thus, we need a toolchain that allows us to systematically reproduce software engineering knowledge in an extended form, e.g. by processing informal input from a domain expert and producing a corresponding IS. In adopting an approach where non-software developers drive the development of ISs, we must ensure that software engineers are still able to support the software development process and achieve more engineering with less intervention. Established practices, such as Model-Driven Software Engineering (MDSE) and traditional source code writing, should remain applicable and efficient, allowing engineers to fine-tune and integrate these systems effectively within existing frameworks.

1.2 Research Questions

The creation of information systems is a challenging task [XL05, ST19]. Model-driven methodologies [BGRW17, FR07] can be used to define transformers that generate software artifacts based on the models provided in a given modeling language. In the context of model-driven generation of information systems, we look at three main layers of architectural abstraction in model-driven architectures [ET12]: *Computation Independent Model*, *Platform Independent Model*, *Platform Specific Model* and the resulting *source code* of the application.

1. **Computation Independent Model**: In the most abstract layer, the system is described informally without stating the implementation details. In this work, the requirements for the system that are defined as natural language are referred to as *Informal Specification*.

   Definition 1 (Computation Independent Model (CIM)). *[BCD+03] A CIM is a model of a system that shows the system in the environment in which it will operate, and thus helps to present exactly what the system is expected to do. It is useful, not only as an aid in understanding a problem, but also as a source of a shared vocabulary for use in other models. In an MDA specification of a system, CIM requirements should be traceable to the PIM and PSM constructs that implement them, and vice versa.*

2. **Platform Independent Model**: A model or a set of models that define the structure of the target application. These models are independent of the concrete platform on which the application is built. We can use these models to formally define the application structure while remaining platform-independent. In this work, we refer to the set of PIMs that define the data structure of the application *Modeled Specification*.

   Definition 2 (Platform Independent Model (PIM)). *[ET12] A platform independent model (PIM) describes a system's structure and functions formally, but without specifying platform-specific implementation details.*

3. **Platform Specific Model**: A model or a set of models that is specific to the technology used in the target platform such as a user interface model that uses specific interfaces of the application front end or contains code snippets.

> **Definition 3** (Platform Specific Model (PSM))**.** *[ET12] A platform-specific model (PSM) includes details that are important to the implementation of a system on a given platform. By platform, MDA means a cohesive set of subsystems and technologies on which a system can execute.*

4. **Application**: The handwritten source code and the generated software artifacts of the application itself. Note that the source code itself can also be seen as a PSM, in the context of this thesis we distinguish between artifacts defined in a modeling language (DSL) to which we refer as PSM and artifacts defined in a general purpose language (GPL) such as Java or Typescript to which we refer as source code or target code, in case it is generated. In the context of this work, we refer to the collection of generated and handwritten software artifacts that compose the final application as the ”Target System”.

Software development typically begins with domain experts describing their problem domain. Thus, we will define a series of transformers that transform informal specifications into application code. As software development is a highly iterative process [GR18, Loo17a] that typically develops systems with an incremental process, adding features over time, we need to take into account that any system we generate needs to be able to be regenerated, adapted, and incrementally refined over time. We can derive the following research questions for a potential transformer that produces an IS:

> **Research Question I**
>
> *What is the structure of a model-driven approach that can be used to build a full-sized information system?*
>
> **RQ1.1** What kind of models are needed to define an information system?
>
> **RQ1.2** How to transform a set of models into a running application?
>
> **RQ1.3** How to allow developers to incrementally transform the application from a prototype to a full-sized real-world system

Domain-specific languages (DSLs) make it easier for domain experts of a particular application area to work with software engineers [FL10] by using higher-level concepts that are focused specifically on their needs [KT08]. This helps both to speak the same language and to address problems more effectively. However, domain expert still needs to have some experience defining models of the given DSL (*e.g.,* knowing the syntax or specific design patterns common to the modeling language). Using LLMs to transform informal textual requirements into a previously defined modeling language, we could bridge the gap between software engineers and domain experts and interpret and formalize any input provided by an individual who has no background in software engineering or system modeling. As LLMs are stochastic models whose output is difficult to predict, we need to evaluate if and to what degree they can be used to produce valid models:

Research Question II

To what degree can large language models be used to transform natural language into a domain model?

RQ2.1 To what degree are LLMs capable of reliably transforming natural language to a model of a given DSL?

RQ2.2 How to use LLMs in an iterative process to incrementally create and modify a domain model?

The models produced by the LLM and the models needed by the generator can be part of different modeling languages (e.g. the LLM produces class diagrams, and the generator needs a combination of class diagrams and GUI-models). The models produced by the LLM are *platform independent models* describing the domain and use case, and the models needed by the generator might be *platform specific models* that describe elements that relate directly to used technologies of the platform such as specific interfaces for GUI or database constraints.

The models produced by an LLM would have to be transformed into the set models needed by the generator. A transformer can be used to perform this task. For data-centric Information Systems in particular, we need a clearly defined set of models that define both the data structure and the user interfaces. In order to ensure data consistency, we merge any data structure model into one singular model, which we refer to as the single underlying model (SUM). Next, we need to investigate to what degree it is possible to derive these models from the SUM. In order not to lose the advantages of agile model-driven software development at this point, it must be ensured that a developer can still adapt the models produced by the M2M transformer.

Research Question III

What methodology is followed to translate platform-independent domain models into platform-specific models for building an information system?

RQ3.1 How can we derive models defining user interfaces of a data-centric information system for efficient data access?

RQ3.2 How can we ensure that the transformation from domain model to application model integrates seamlessly into the agile-iterative development process and allows for the customization of the generated models?

Up to this point, the research questions target primarily the individual transformations and generation steps within a tool chain. However, we also have to consider the requirements for the tool chain as a whole. As we define a generative approach, we need to be able to modify the elements that are generated. This applies not only to generated source code, but also to generated models. As we cannot anticipate all use cases that might occur, we need to ensure that the tool chain is flexible enough to be applied in different domains. In addition, we need to ensure a minimal degree of usability of the generated application. Hence, we can define the fourth research question:

Research Question IV

How to ensure adaptability, variability, and usability throughout a generative tool chain for information systems.

RQ4.1 How to ensure adaptability and variability of generated models and target code?

RQ4.2 How to define the tool chain for an information system so that it is customizable to any target domain?

RQ4.3 What UI-elements are required to ensure the usability of the generated information system?

Having defined our research questions, we now can develop a corresponding research approach. These questions are vital because they shape the direction of our study. They function as a road map, outlining the destination and the path to reach it.

1.3 Contribution

In this work, we define a tool chain consisting of three major transformers, as shown in Figure 1.2. The tool chain is capable of transforming informal specifications into a domain model, which is transformed into a set of platform-specific models that, in turn, can be used by a generator framework to generate an IS. The capabilities of the generator framework and transformers were established in multiple real-world projects, notably the MaCoCo project which serves an active user base of more than 200 daily users (at the time of writing). We were able to show that LLMs could be used to define valid models for several MontiCore-based DSLs [HKR21] and could be integrated into the proposed tool chain to iteratively define and modify a single underlying model, which serves as a foundation to generate the IS.

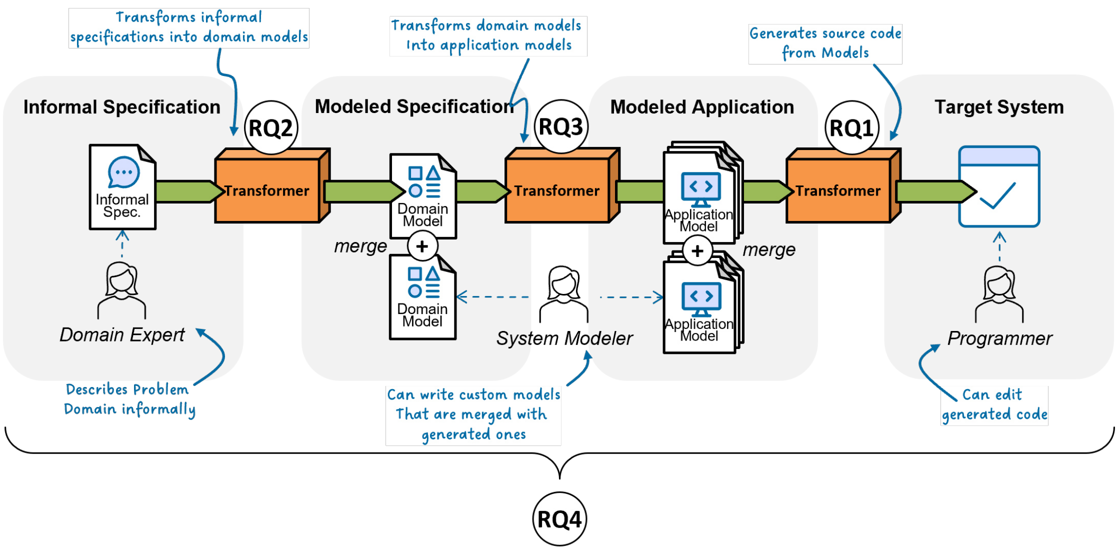


Figure 1.2: Tool chain Overview: Transforming natural language first into *Domain Models*, next into *Application Models*, and finally into a *Target System*

1.4 Requirements and Objectives

We are guided by the research questions to define three transformers. We propose further overarching requirements that the tool chain needs to fulfill, to create a robust and flexible system that can be applied in real world systems. We place particular emphasis on:

	TRL	Description
	9	Actual system proven in operational environment
Deployment	**8**	System complete and qualified
	7	System prototype demonstration in operational environment
	6	Technology demonstrated in relevant environment
Development	**5**	Technology validated in relevant environment
	4	Technology validated in lab
	3	Experimental proof of concept
Research	**2**	Technology concept formulated
	1	Basic principles observed

Table 1.1: Technology readiness levels (TRLs) as defined by NASA [Com14]

- **Technology Readiness Levels:** Initially create a prototype that can transition to a full-size application. We use TRLs to measure the readiness of the produced application.
- **Adaptability:** Artifacts that are iteratively generated by the tool chain must be modifiable.
- **Variability:** Ability of a software system to be efficiently extended, changed, customized, or configured for use in a particular context.
- **Usability:** Both the initial use case definition and the resulting information system should be easy to use for domain expert.

1.4.1 Transition to Real-World Systems

In software development, applications can be experimental, designed for lab testing, or full-scale for real-world use. Real-world applications are typically operated by users without specialized training, necessitating high technological robustness to meet both functional and various nonfunctional requirements. The technology readiness levels (TRLs) of NASA [Com14] are a good metric to measure the gap between experimental software and a finished product (see Table 1.1). In a university environment, the target readiness level often does not exceed TRL-3 (research). ESA also provides a refined version of these TRLs (*cf.* [G$^+$13]).

TRLs adapted for Software Projects by ESA (ISO 16290:2013): Software developed in a research context is typically used as a proof of concept and to analyze methodologies or algorithms. Within the fast development cycles of agile software development, this level is maintainable with a small group of relatively inexperienced software developers. However, TRL-3 is not suitable for software deployment in a real-world scenario. Extra

Level	**Description**
TRL 1	Basic principles observed and reported: Initial observations and descriptions of the software concept are made.
TRL 2	Technology concept and/or application formulated: The concept is further developed, and potential applications are identified.
TRL 3	Analytical and experimental critical function and / or Characteristic Proof of Concept: The software concept undergoes analytical and experimental studies to validate its potential functionalities.
TRL 4	Breadboard validation in laboratory environment: A prototype (or breadboard) version of the software is developed and tested in a lab setting.
TRL 5	Breadboard validation in relevant environment: The prototype software is tested in a relevant environment, closer to where it would actually be used.
TRL 6	Model or prototype demonstration in a relevant environment: A more refined prototype or model of the software is demonstrated in a relevant environment.
TRL 7	System prototype demonstration in an operational environment: The software prototype is demonstrated in an operational setting, showing its functionality in a real-world scenario.
TRL 8	Actual system completed and qualified through tests and demonstrations: The software system has been finalized, tested, and qualified for operational use.
TRL 9	Actual system proven through successful mission operations: The software has been used successfully in real mission operations, proving its reliability and effectiveness.

Table 1.2: Technology Readiness Levels (TRLs) [HJ16]

effort and additional methodologies must be invested to raise TRL-3 to TRL-9 (*cf.* Chapter 8). The building blocks and tools used and referred to in this work can also be assigned to the corresponding TRL (*cf.* Table 1.3).

R1 (Transition Capability): The framework must support the developer to transition (parts of) the application from a prototype state (TRL 3) to a production version (TRL 9) of the application.

1.4.2 Adaptability

Within this work, we refer to adaptability as follows:

TRL	Project Name	Type	Description
9	MontiCore	Tool	Language Workbench (*cf.* Section 3.3)
6	LLM4CD	Building Block	Transformer (*cf.* Chapter 4)
4	CD2GUI	Building Block	Transformer (*cf.*.Chapter 6)
6	MontiGem	Building Block	Generator (*cf.* Chapter 7)
9	MaCoCo	Tool	Web Application (*cf.* Chapter 8)
7	ADD	Tool	Web Application (*cf.* Section 9.1)

Table 1.3: Technology readiness level mapping for building blocks and tools, in the December 2023 assessment.

Definition 4 (Adaptability)**.** *[SC01] The term "adaptation" in computer science refers to a process where an interactive system (adaptive system) adapts its behavior to individual users based on information acquired about its user(s) and its environment.*

This approach quickly provides domain expert with a web application for the given specifications. Although this enables a quick start to a running prototype, adaptability must be considered, especially with respect to any user interfaces that should be synthesized based on the initial input of the domain expert. It is expected that not all elements of the single underlying model should be represented with the same structured user interfaces. Reasons to deviate from the default generation process for a specific element in the domain model can be:

- **Privacy Concerns:** Data fields such as passwords, test results, contact information, and corporate data, should not be visible to anyone and should not be transmitted to the user without extra caution.
- **Derived or computed data:** Data fields such as checksums, last edit time, cached values for efficiency optimization, and data that contain the results of computations and should not be modified by the user. If unrestricted access to the database is enabled, the user could disable his own account or any other by editing the password hashes.
- **Aesthetics or Readability:** Unreadable fields that are hard to visualize such as JSON-objects or floating-point numbers with a high decimal count. Some fields are not optimized to be displayed in a cell of a table, such as lists or complex objects. These should be omitted from the default view and displayed in a custom or adapted table.
- **Relevance:** Fields that are irrelevant for day-to-day use or certain tasks, such as unique database IDs or revision information, should be hidden from the user. The

data structure has to be able to store the information needed for any use case, but very often the typical use case only requires the interaction with only a fraction of the data set. Thus, many fields can be omitted for a user interface that is meant to be used on a daily basis.

- **Domain specific presentation:** Field can be presented differently based on the domain they are presented in. For example, temperature can be presented differently than an interest rate, a weight, or a maximum payload.

For the reasons mentioned above, we need to introduce mechanisms to adapt any user interfaces we derive from any single underlying model. Similarly, we cannot rely on the semantics of the given model to automatically infer these differences, as they may change from use case to use case, for example, because privacy policies are different in different projects. It is also impossible to anticipate any future use cases.

In generator frameworks for web applications, all code created can be extended using inheritance [HR17], with the exception of HTML files, as this markup language does not support inheritance. This limits the developer in two important aspects. Firstly, by not being able to adapt the generated HTML-Code he can not add further elements to the user interface and is restricted to adaptation in the business logic in either front end or back end. Secondly, because the developer edits the target code, he has to make adjustments to his code after each change to the data structure. As long as the corresponding models are customizable by the developer, this issue does not pose a problem. Thus, we can conclude the following requirement:

R2 (Adaptability): All derived models and the generated code must be adaptable by the developer to respond to the use case-specific properties of the class diagram.

These requirements define a strong adaptability for the transformation approach. However, this high degree is necessary because we not only target the generation of a prototype system, but also aim for a prototype that can transition to a full-sized real-world system.

1.4.3 Variability

With this approach, the developer benefits from many automatically generated artifacts. However, in order to maintain pragmatism, the developer must also be able to adapt the sum of the artifacts to his use case. The concepts of variability offer a suitable approach that we would like to apply. Galster et. al. define Variability within the context of software development as follows:

Definition 5 (Variability [GWT+13, VGBS01, GWT+14]). *Variability is commonly understood as the ability of a software system or software artifact (e.g., component) to be changed so that it fits a specific context. Variability allows us to adapt the structure, behavior, or underlying processes of the software. These adaptations are enabled through [predetermined] variation points and variants as options that can be selected at these variation points.*

Adaptability refers to a software system's ability to adapt to changing requirements or environments without major changes to its structure or functionality. Key aspects include flexibility, robustness, and maintainability. Variability refers to a software system's ability to support different configurations or customizations based on predefined options. It is often used in software product lines. Key aspects include feature modeling, configuration management, and reuse. In short: Adaptability deals with handling changes, while variability focuses on supporting various configurations or customizations. Reasons to handle variability can be:

1. **Integration of Domain Specific GUI Components** Although any UI modeling language used might offer a wide range of GUI-Components, it is very likely that there are use cases where components are needed that are not yet provided by the DSL. Therefore, we need a mechanism to systematically integrate custom components.

2. **Generic Component Modification** The provided GUI components might not provide the desired functionality required by the developer of the application. It is conceivable that an adapted set of information or functions should be systematically integrated into the user interfaces.

Variability and adaptability face similar challenges and requirements, so the concerns mentioned in Section 1.4.2: Privacy concerns, derived or computed data, aesthetics or readability, and relevance apply here as well.

1.4.4 Usability

The approach presented is intended to improve the existing software development methodology. For it to be successful, it needs to be more efficient in producing web applications than common approaches. Model-driven approaches typically reduce the amount of handwritten code by providing generated source code, but the effort saved can be canceled out if the developer has to make time-consuming adjustments to the generated code afterward. Thus, we need to maximize both the portion of code we can generate and to focus on mechanisms that help the developer to make adjustments to generated code.

We identify two user groups that are affected: (1) End users of the web application must have a system developed with a focus on user experience. (2) Developers using the generator framework must have a system that is easy to use.

R3 (Usability): The framework must support the domain expert in providing an efficient, intuitive, web information system to access and manage data.

Usability of the Target Application

The usability of websites is not only an interesting research topic, but also a highly profitable commercial topic. Therefore a lot of tools exist to analyze the various aspects of a web application. According to Kumar and Hasteer [KH17], the usability of a website can be defined by the following metrics:

- **Navigability** Defines how well a website is frequented by users, how long they remain on the page, and to what extent they move between pages. The accessibility can be measured using the following attributes: *Returning Visitors*, *Page Views*, *Pages per Session* (how many pages were visited during one session), *bounce rate* (how many visitors leave the page without navigating to any second page on the website), *Session duration*, *Average Drop-Off per interaction* (likeliness to leave the page after any interaction). Google Analytics[1] or others can be used to measure the above attributes.
- **Readability** Defines how easy or difficult a text is to read. It is typically measured in scores. The Readable.io tool[2] [TMR+23, MEBF+19] provides spatial measures: A generic *rating* from A to F and an *Avarage grade level* comparing the readability with the formal education needed to understand the text (e.g., fifth grade student). There is also the more popular *Flesh Kincaid* measure [Fle79], scoring the readability from 0 (hard to read) to 100 (easy to read).
- **Loading Speed** Defines the time required to fully display the web page. Different institutions provide different standards here. Google defines the loading speed for the first contentful paint (FCP) as 'Good' if the page loads within the first 1800ms, as 'Needs Improvement' if it loads within the first 3000ms, and as 'Poor' if the page takes longer to load [HŠB21].
- **Accessibility** Defines the practice of making your websites usable by as many people as possible, considering not only people with disabilities but also the variety of devices e.g. mobile or wearables that might want to access the website [MBLK23].

[1] analytics.google.com
[2] readable.io

The tool Qualidator[3] [KJN19] shows possible weaknesses of the site and gives three scores in addition to the specific problems of the site. It provides an overall accessibility score, a usability score, and a search engine optimization score.

- **Functional Performance** Defines how well the page works and how well it is received by the end user. Nibbler[4] [KH17, JP21] is a tool to analyze these aspects of a website, based on multiple metrics such as *server behavior* checking for basic functions such as 404 pages and well-definedness of links. Further metrics are *Marketing*, whether there are analytics data, a linked X[5] (former Twitter[6]) account, and the freshness of content and *Technology* analyzing printability of pages, the mobile capability, and in general how well designed and built the website.

With minor exceptions (e.g., marketing measures with social media), these metrics provide a good basis on how to measure the usability of a generated web information system [CCOTF09] such as the target application of the generator framework that is presented in this thesis.

Usability of the Generator framework

In the previous section, we provided measures on how to test whether the requirements for a highly usable website are met. Now we need to provide requirements for the framework to enable the developer to create such a website. In contrast to usability measures for web applications, there are fewer tools available to measure the usability of a generator framework. In this thesis, we refer to the usability of the generator as its ability to relieve the developer of work. Thus we can derive a few requirements for model-driven tooling:

- **Flexibility:** Because the adaptation of a generator to a new target domain often happens, it's important to prioritize flexibility to ensure that it can be integrated and reused across various projects. This flexibility minimizes the learning curve and time investment required to familiarize yourself with the system, making it more efficient for developers to adapt the framework to different needs.

- **Customizability:** To maintain efficiency and avoid clutter, a framework should allow easy addition and removal of extensions. This ensures that each project includes only necessary components, preventing the accumulation of unused elements. By tailoring the framework to the specific needs of each project, developers can keep their workspaces streamlined and focused on relevant tasks.

[3] www.qualidator.com
[4] www.nibbler.silktide.com
[5] www.x.com
[6] www.twitter.com

- **Iterativity:** The framework should be designed to generate code in stages, building progressively over each phase, and increasing the generated code incrementally. This step-by-step approach ensures a structured development process and facilitates easier tracking of changes and additions.
- **Robustness:** The generator should notify the developer of any issues (such as type checks and context conditions) during generation time and avoid errors that come up during run time.
- **Ease of use and documentation:** To ensure developers can easily start using the framework, there should be minimal initial obstacles. Comprehensive documentation and introductory guides are essential to facilitate this.

1.5 Thesis Structure

The structure of the thesis is outlined as follows:

Introduction (Chapter 1) The first chapter provides an introduction to the topic, as well as a presentation of the research questions and requirements of the proposed tool chain.

Methodology (Chapter 2) The second chapter presents the methodology and gives a short introduction to each transformer.

Foundation (Chapter 3) This chapter gives a provision of the necessary foundations, including the basic architectures of web applications, the foundations of model-driven software development, the language workbench MontiCore and preexisting domain-specific languages.

LLM4CD (Chapter 4) The fourth chapter presents the modeling capabilities of large language models and introduces the first transformer (LLM4CD) used to transform natural language into CD4A and other languages developed for web application generation.

DSLs (Chapter 5) The fifth chapter introduces both GUIDSL v1 and GUIDSL v2 as the primary languages used to define user interfaces for information systems. In addition, the usage of the Tagging Language for Class Diagrams is presented.

CD2GUI (Chapter 6) This chapter presents the next transformer CD2GUI and its options for reconfiguration. It also provides insights into mechanisms for adaptability and variability of CD2GUI, as well as several extensions to increase usability of the information system.

Generator (Chapter 7) Chapter seven focuses on the final transformation using the MontiGem framework to generate Typescript, HTML and Java code. We present the architecture of the target application and introduce the artifacts generated. The key principles of front-end and back-end generation are presented, as well as the generated command pattern to enable seamless data transfer between the server and the client.

MaCoCo (Chapter 8) This chapter presents MaCoCo, a case study on a full-size web application created using the MontiGem-framework.

Examples (Chapter 9) Chapter nine presents additional use cases of the model-driven approach, highlighting lessons learned from each project.

Discussion (Chapter 10) This chapter presents a discussion of the thesis and its results, as well as a contextual comparison.

Conclusion (Chapter 11) The final chapter provides a conclusion of the thesis with final thoughts.

1.6 Publications

This thesis is based on several years of research. Thus, some of the findings have been previously published in various contexts prior to the completion of this thesis. As a result, certain results, figures, data, and other content within this thesis have already appeared in conferences and journals, or are currently in press or in preparation. This section offers an overview of these publications.

- [ANV+18] describes the initial concept of the usage of the MontiGem generator framework (Chapter 7), to generate information systems based on a limited set of modeling languages. The Alpha version of MontiGem was developed in collaboration with Simon Varga and Kai Adam. The concepts were continued in later versions of MontiGem. MontiGem was refined in several Student and Industry projects. The paper was a joint effort of the authors.

- [GHK+20] focuses on the effects of simultaneous generation of the front- and back-end using MontiGem as an example. It specifically focuses on the role of the domain expert in defining the information system through models. The results are presented in more detail in Chapter 7.

- [AMN+20] describes the lessons learned from the early development and application of the MontiGem generator framework to generate enterprise information systems (*cf.* Section 7.3.1, Section 7.3.1).

- [GMN+20] presents a methodology that enables the transition (*cf.* Section 4.5.5) from a prototype implementation to a full-size real-world system in the context of model-driven software engineering.
- [DGM+21] describes a process on how to retrofit a generative approach to an existing implementation (*cf.* Section 8.2). This paper is based on insights from the MaCoCo project.
- [BGK+23b] Summarizes lessons learned from the application of MontiGem in five different domains. The author of this thesis contributed the foundation and the section describing MaCoCo, *cf.* Chapter 8. Christian Kirchhof contributed the section describing the 'Ford' project (*cf.* Section 9.3). Arkadii Gerasiomiv contributed the section describing the System for energy management, and Sebastian Stüber contributed the section describing the invidas (*cf.* Section 9.2) project.
- [NGM+24] Presents the usage of model-driven methodologies to develop the full-size real-world system MaCoCo. It summarizes the lessons learned from the development process of the MaCoCo project (*cf.* Chapter 8).
- [Kro23] shows an implementation of the concepts to extend the GUIDSL v2 for adaptability as previously defined in [GMN+20] (*cf.* Section 4.5.5).
- [Sla22] examines multiple variants of generated user interfaces for website navigation, which are aimed at enterprise information systems and are based on the provided class diagram (*cf.* Section 6.3.2).
- [Ort22] evaluates the usage of tagged class diagrams (*cf.* Section 3.5, Section 6.5.1) in order to automatically introduce model-driven role-based access control into the generated web application.
- [Fel23] performs a user study to evaluate the ability of MontiGem to operate as a low-code development platform compared to commercial software system development solutions (*cf.* Section 9.4).

Chapter 2

Model-Driven Method

We address the three research questions and subsequent requirements by proposing a methodology and a corresponding tool chain that enables the iterative creation of a web-based information system using informal specifications as input.

Contents

2.1 Roles

Individuals that can interact with this tool chain and its results can assume four roles:

a) The *domain expert*: An individual, also known as a subject matter expert, who has knowledge of a specific domain (*e.g.,* finance or healthcare). The domain expert is one of the stakeholders that defines the use case of the targeted application and describes it in natural language at the beginning of the generation process of the target application. Within this methodology, domain expert is not required to have any experience in the definition of any models or is required to have any programming skills. The domain expert has both functional and non-functional requirements at the target application and is able to evaluate the resulting application against his input.

b) The *system modeler* has expertise in defining models for the languages used in this tool chain. He is familiar with the modeling languages that are used in this tool chain and can define, extend, and modify models accordingly. He does not have to be an expert on the subject matter and is not required to have any programming skills.

c) The *programmer* is familiar with the target languages (HTML, TypeScript, and Java), but does not need to have any subject matter expertise nor is required to know any of the modeling languages used. The programmer can extend and modify the target application directly. By adding handwritten code directly to the generated source code of the application.

d) The *Toolsmith*, develops and configures the individual tools within the tool chain.

e) The *user* is the individual who interacts with the produced application.

Within this tool chain, an individual can have multiple roles as they are not exclusive. It is very likely that the system modeler is the same person as the programmer or that the user is the same individual as domain expert.

2.2 Transforming Natural Language into a Web Application

The proposed tool chain can be broken down into three major transformers:

① Natural language to Class Diagram transformer (LLM4CD)

② Class Diagram to Graphical User Interface Model Transformer (CD2GUI)

③ MontiCore-based Generator for Enterprise Management Systems (MontiGem)

They operate as follows:

2.2.1 Transformer overview: LLM4CD

We start by interpreting and formalizing the informal input provided by domain expert (*cf.* Figure 2.1). As the targeted information system is data-centric, the first transformation converts the user input into a class diagram. This class diagram ideally describes most of the relevant data concepts of the application. The input of domain expert can be changed iteratively, to adapt the application to changing requirements, as LLM4CD is the first transformation, there are no dependencies of these specifications on any preceding models. The domain expert can freely choose his input. These modifications can also be provided as informal specifications by the domain expert (Chapter 4). LLM4CD is well suited to interpret informal specifications and to produce a corresponding class diagram. However, in case a very specific class diagram is already envisioned, the user would have to specify this model very precisely. In this case, it would be more efficient to enable the direct definition of a class diagram. Therefore, an additional mechanism is used to include class diagrams directly. *CDMerge* [LRSS23], a tool to merge two class diagrams and detect possible conflicts within that merge, is used to add additional classes to the output of LLM4CD. Thus, allowing for a combination of informal specifications

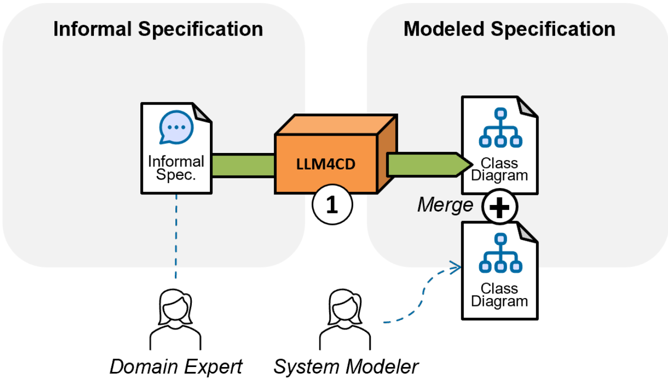


Figure 2.1: Transforming informal specifications into class diagrams: LLM4CD leverages the NLP capabilities of LLMs to transform continuous text into valid CD4A syntax. In order to permit the inclusion of preexisting models, a *System Modeler* can merge handwritten models with any class diagram that was produced by the LLM.

and formal data structures. This allows the domain expert to vaguely describe certain aspects of the application while allowing a system modeler to precisely define specific elements, such as classes for the run-time environment or API descriptions. As this model is the basis for all subsequent transformations, we refer to the merged model as the root class diagram. The first transformer (LLM4CD) is used to transform *domain descriptions* into platform-independent *domain models*. LLM4CD uses large language models that are based on statistical analysis. Therefore, there is a chance that it can return an invalid model (e.g. bad syntax, or semantically wrong). In this case, domain expert can simply run the transformation again to produce a valid model. Another aspect of large language models is the unpredictability of the returned output, minor changes in input can result in major changes in output. We address the challenges of LLM-based algorithms in Chapter 4.

2.2.2 Transformer overview: CD2GUI

CD2GUI uses the merged root class diagram as input and derives for each class a set of GUI-models (*cf.* Figure 2.2). Each describes a web page in the targeted web application. The modeled user interfaces allow the user to create, read, update, and delete (CRUD) data defined by root class diagram. Depending on its configuration, additional user interfaces can be produced that are based on root class diagram, such as change data capture (history), access control, or class-specific dashboards. Similarly to LLM4CD, we need to provide adaptability to the models produced. CD2GUI can be configured to produce GUI-models differently based on a class, association, or attribute being pro-

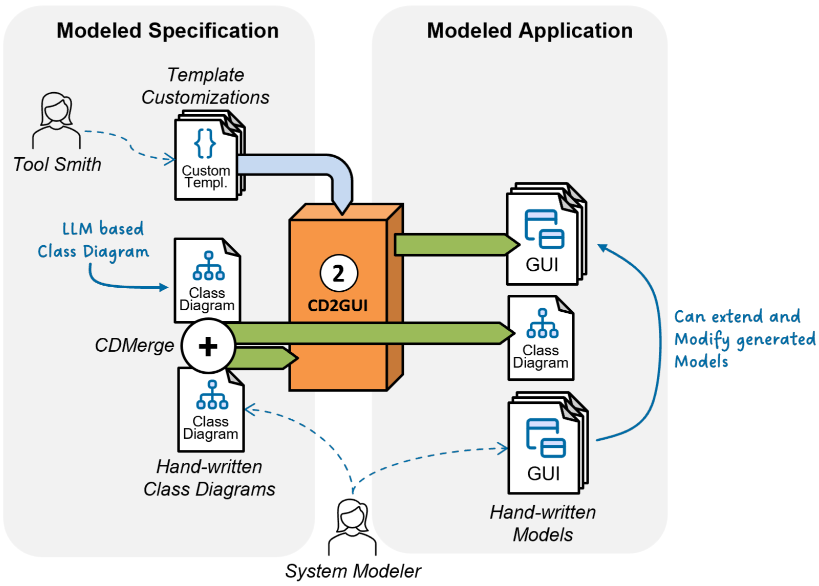


Figure 2.2: Transforming data structure models into user interface models: CD2GUI provides the necessary user interface needed to create an application. It can be configured with Custom Templates (*cf.* Section 6.6.1), and receives Class CD4A diagrams as input. Within the tool chain, LLM-based CD4A models can be merged with handwritten ones in a pre-processing step. CD2GUI produces GUI-models and hand over the merged CD to the next tool in the tool chain.

cessed (Section 6.6). A system modeler can change the behavior of the generator based on the type of data handled. This mechanism can be used to include specific visualizations, functions, and behaviors in the user interface each time a specific type is handled, e.g., displaying a temperature with a gauge or hiding private user data in all user interfaces. A second mechanism for adapting models is introduced into the modeling language used to define GUI-models (GUIDSL) itself. Hand-written models can be used to adapt generated ones, allowing modification of the CD2GUI output. Each element in a GUI-model can be referenced and modified, allowing removal, replacement, and addition of any element defined in a GUI-model.

The single underlying model is used to create GUI-models, but it also passes through the next transformation step unchanged. CD2GUI is used to transform domain models that primarily describe the target domain into application models that can be used to define software for that target domain.

2.2.3 Transformer overview: MontiGem

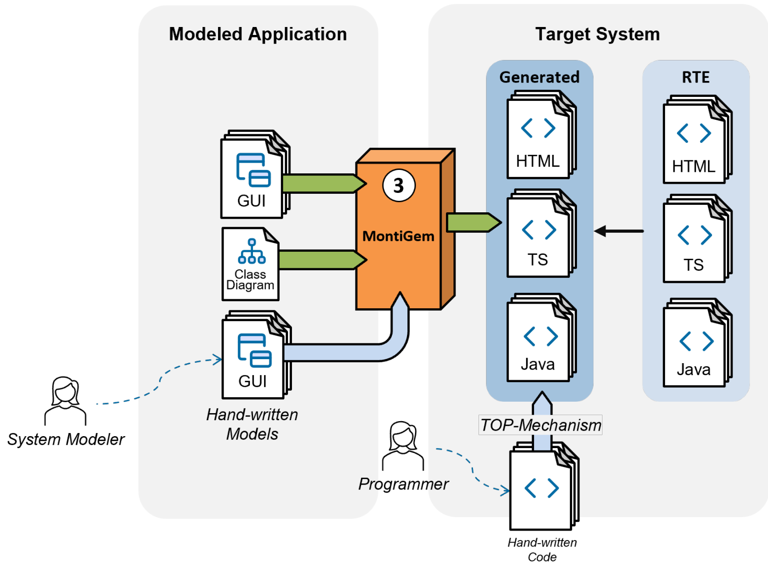


Figure 2.3: Transforming system models to the target system (web application). A system modeler can contribute additional models, a programmer can add hand-written code. (Excerpt from Figure 2.4)

MontiGem is used to transform a set of models into a web application (*cf.* Figure 2.3). MontiGem follows the basic architecture of a generator as described in [HKR21]. As all DSLs used in this approach are based on the same language workbench, we use the tools provided by that workbench to develop the corresponding generator. In order to generate a web application MontiGem requires at least a set of class diagrams and GUI-models. In addition, further models can be provided, such as tagging and OCL models. In addition to the GUI-models provided by CD2GUI, a system modeler can provide additional GUI-models, defining further custom pages. MontiGem produces a fully functional data-centric web application, consisting of a presentation layer, an application layer, and a persistence layer. The generator provides both the server and the client with the root class diagram-data structure and a corresponding command pattern that is used to handle data transfer between both. The server is provided with a persistence API to define and access the data types declared in the root class diagram. In addition to the core functionalities MontiGem also provides several additional classes that help the developer create a sophisticated application. The application produced by MontiGem can be used as is. If necessary, a programmer can modify and extend the code produced by MontiGem using the TOP-Mechanism ([HR17]). Although the above method describes many options to configure, adapt, and modify the generation process, it is still capable of operating on the basis of the initial informal specifications of domain expert alone (RQ 1). The method is intended to be used iteratively, enabling the transformation from a simple unconfigured prototype to a full-size real-world system (RQ 3) such as MaCoCo (*cf.* Chapter 8).

2.3 Tool Chain Overview

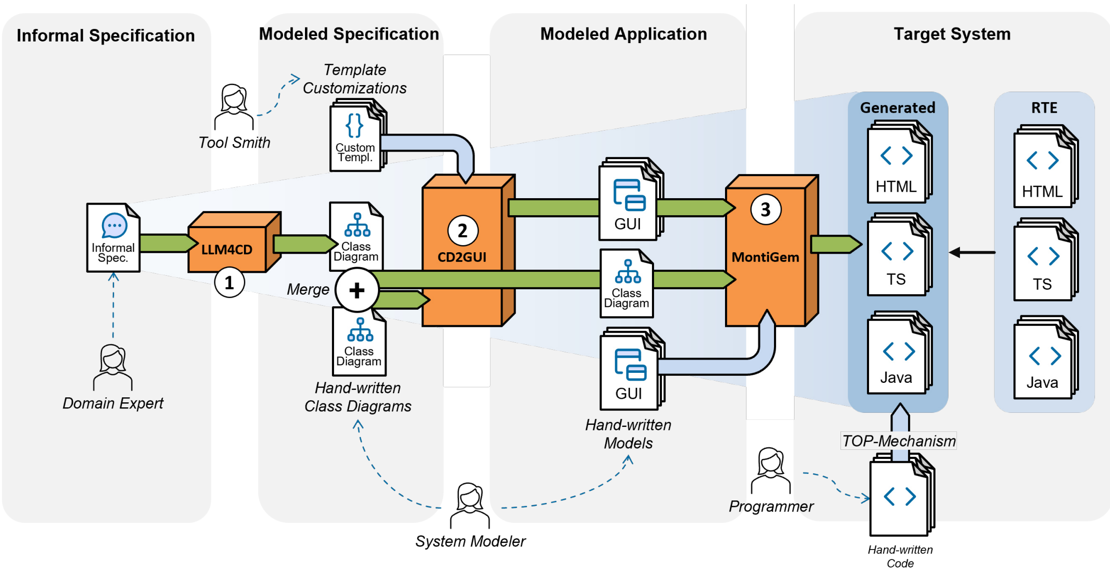


Figure 2.4: Method Overview: Transforming informal specifications into a fully functional web application

The development of web applications is complex [GM01, CDDM09, XL05] and costly [RJW03]. It starts from an informal concept and evolves through multiple iterations, involving various skilled individuals, into a complete system. The key is adherence to the requirements of the stakeholders [FR07]. By combining these three transformers, this thesis introduces a method to generate web applications from informal specifications, a complete overview is shown in Figure 2.4. This process can produce a prototype from an informal specification. A domain expert provides informational requirements, which are transformed by a transformer (1 LLM4CD) using a large language model (LLM) into domain models [BGS05]. This model can be merged with further models defined by a system modeler. The resulting model is the *single underlying model* used to define the web application. The system modeler can also define custom templates to configure the next transformer (2 CD2GUI) The single underlying model is then converted into application models and further transformed into the *Target System*'s source code (3

MontiGem).

Running Example

The following example illustrates the approach (*cf.* Figure 2.4): The domain expert defines *"A web store for books"* and passes the informal specifications to the first transformer (1 LLM4CD), this informal specification is transformed to the class diagram shown in Figure 2.5. The model can be reviewed and modified iteratively using informal specifications in LLM4CD to adjust the model. The domain expert could add the requirement to enable the bookstore to also offer videos. This would result in LLM4CD adding a `Video`-class next to the `Book`-class without requiring domain expert to perform this change directly on the model. LLM4CD sends the class diagram to the next transformer: 2 CD2GUI. It produces for each class (`BookStore`, `Book`, `Order`, `Customer`) two GUI-models that define two kinds of pages: An overview page to find individual objects, *e.g.,* the overview page for the *Book*-Class, which lists all books in the database, a details page, used to inspect individual objects, *e.g.,* to change details on a specific `Order`. In addition, more GUI-models are produced to improve the overall user experience. CD2GUI passes the input class diagram and GUI-models over to the next transformer 3 MontiGem. MontiGem produces Java, HTML and TS files and generates a web application based on the input models. Thus, a basic prototype of a Web application is produced for a bookstore, based on the initial input by the domain expert. As the developed application is only a prototype, the system modeler and the programmer can iteratively edit and refine any of the produced models and source code in order to iteratively transform the prototype into a real-world application.

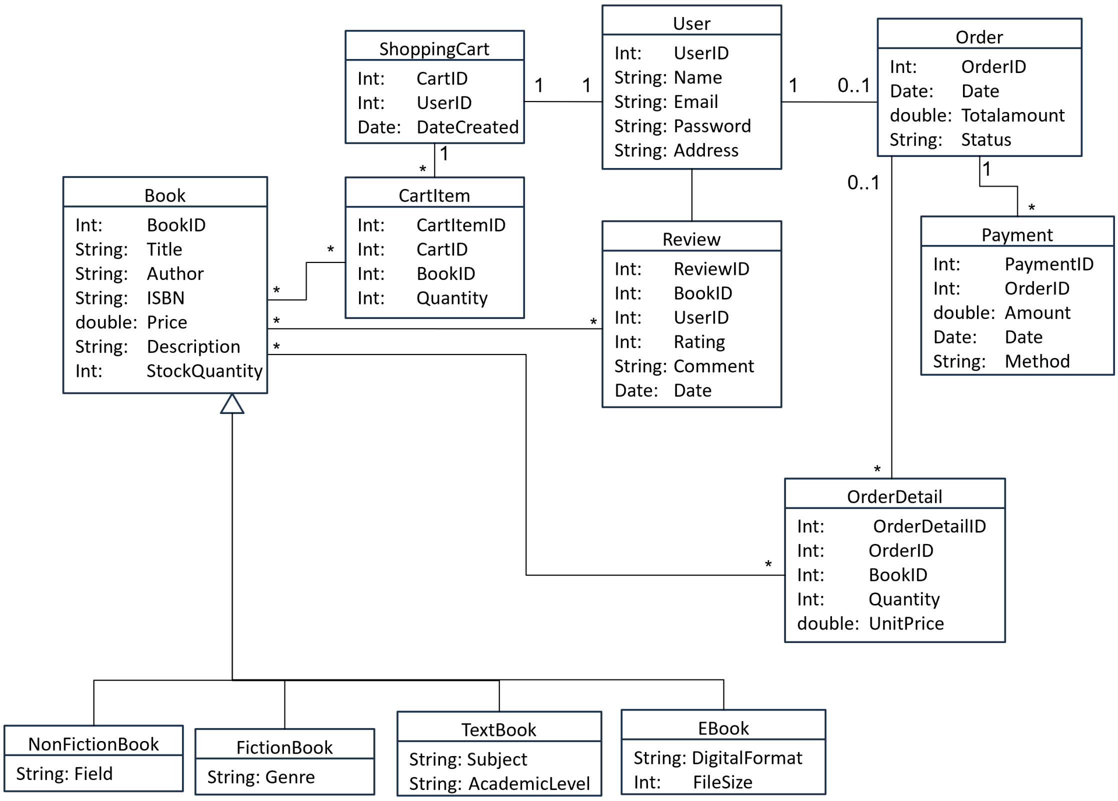


Figure 2.5: Visualization of a simplified class diagram, produced by LLM4CD based on the input *"A web store for Books"*.

Chapter 3

Foundations

In this chapter, we introduce basic concepts and foundations relevant to this thesis:

Contents

3.1 Web Application Architectures

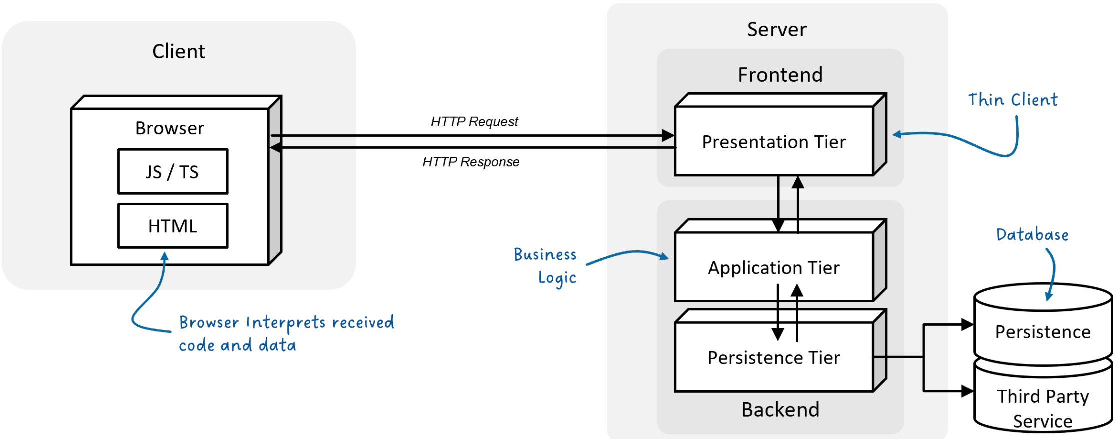


Figure 3.1: Typical architecture of a web application

One goal of this thesis is the generation of web applications; therefore, we will have a closer look at the structure of such a system and what its basic components are. Figure 3.1 depicts a typical architecture of a web application. A web application consists of two sides: The *client* located on the user system and a *server* located on the application provider system. On the client side, the user uses a browser to request data from the server. Upon request, the server provides the client with runnable code (e.g., HTML, Typescript, or PHP) that is executed in the *browser*, providing the user with an interactive user interface. The executable layer that presents the data to the user is defined in the *presentation tier* of the *server*. This layer is used to visualize data input and output to the user and enable tailored interactions with the server to the specific domain or use case at hand. A user always interacts through the *front end* with the server. Next to the *front end* is the *back end*. It contains both *Application Tier* and *Data Tier*. The Application Tier defines any business logic needed to run the application. Any logic that is needed to process data inputs that are received through the Presentation Tier is defined in this layer. Both the *presentation tier* and the Application Tier rely on a set of data. This data is managed in the *persistence tier*, which defines how the data is stored and who can access it. Thus any data storage (*persistence*) and external data sources (*third party service*) are linked to the Data Tier. The web application that is generated by MontiGem follows this basic architecture for its target application.

3.2 Model-Driven Software-Development

Model-Driven Software Development (MDSD or MDD) is a software engineering approach that emphasizes the use of models to design, develop and maintain software systems. The main goal of MDD is to improve the productivity, quality, and maintainability of software by automating repetitive tasks and providing a higher level of abstraction when working with complex systems. In MDD, models are used as primary artifacts to capture the essential elements of the system being developed, such as structure, behavior, and requirements. These models are often created using specialized modeling languages like Unified Modeling Language (UML) [Obj17], SysML [H$^+$06], or Domain Specific Languages. The development process in MDD involves the following steps:

1. **Requirements Analysis and Modeling:** The requirements of the system are captured and represented using suitable models.
2. **Design and Architecture:** High-level system design and architecture are defined using models, which serve as a blueprint for the system.
3. **Model Transformation:** Models are transformed into other models or lower-level artifacts, such as source code, configuration files, or database schemas, using automated tools and techniques such as model-to-model (M2M) and model-to-text (M2T) transformations.
4. **Implementation:** The generated code and other artifacts are integrated, tested, and deployed.
5. **Maintenance and Evolution:** Changes and enhancements to the system are made by modifying and updating models or integrating handwritten artifacts, which are then transformed to update the corresponding artifacts.

MDD offers several advantages, such as increased productivity due to automation, improved quality through better abstraction and consistency, and easier maintainability by focusing on models rather than code. However, it also has some drawbacks, such as the need for expertise in modeling languages and the potential for a steep learning curve for novice practitioners.

3.3 MontiCore

In this chapter, we introduce the key concepts behind MontiCore. The complete documentation of the latest version of MontiCore can be found at `monticore.github.io` [HKR21]

In this work, we use many languages based on MontiCore. In addition to that MontiCore was used to generate a lot of the infrastructure that is used to work with those DSLs.

3.3.1 Overview

MontiCore [GKR+06, KRV08, GKR+08, HR17, HKR21] is a language workbench that supports the development of domain-specific modeling languages and code generation. A language workbench, in general, is a software development tool that allows users to create, modify, and analyze DSLs to simplify the development process and increase productivity. Some popular language workbenches include JetBrains[1] MPS, Xtext, and Spoofax[2]. MontiCore provides a framework to design and implement DSLs tailored for specific tasks and domains. MontiCore enables the creation of modeling languages by defining the context-free grammar for a particular domain. It then provides infrastructure for the development of generators (Figure 3.2) for the synthesis of code from models created in the defined DSL. This allows for a higher level of abstraction in software development, making it easier to specify complex systems and generate software artifacts (such as code, documentation, or configuration files) from the models. MontiCore has several features that make it a powerful tool for DSL development:

1. **A Modular Language Development Approach:** MontiCore supports the creation of modular and reusable language components, allowing developers to build languages tailored to specific domains and applications [HJK+23, JR23, KRV10, Völ11].

2. **Code Generation:** The MontiCore system provides the means to generate code in various programming languages, based on the defined DSL models.

3. **Model Analysis and Transformation:** Support of the analysis and transformation of models, enabling the creation of model-to-model and model-to-text transformations.

4. **Extensibility:** The language workbench can be extended with additional features and plugins to cater to specific requirements.

Within MontiCore, a language is defined with context-free grammar. Based on that grammar a parser, a data structure for an abstract syntax tree (AST), and the implementation for a visitor pattern [GHJ+95] to traverse the given AST can be generated. Typically the AST is extended by a Symbol Table (ST) mapping uniquely identifiable names and nodes in the AST to a collection of scoped symbols. The symbol table can be used to define and check context conditions (CoCos). Before transforming the AST into target code, one or more model-to-model transformations can be applied to extend or modify the input AST. These can be implemented by hand as in this work or be

[1] https://www.jetbrains.com/
[2] https://spoofax.dev/

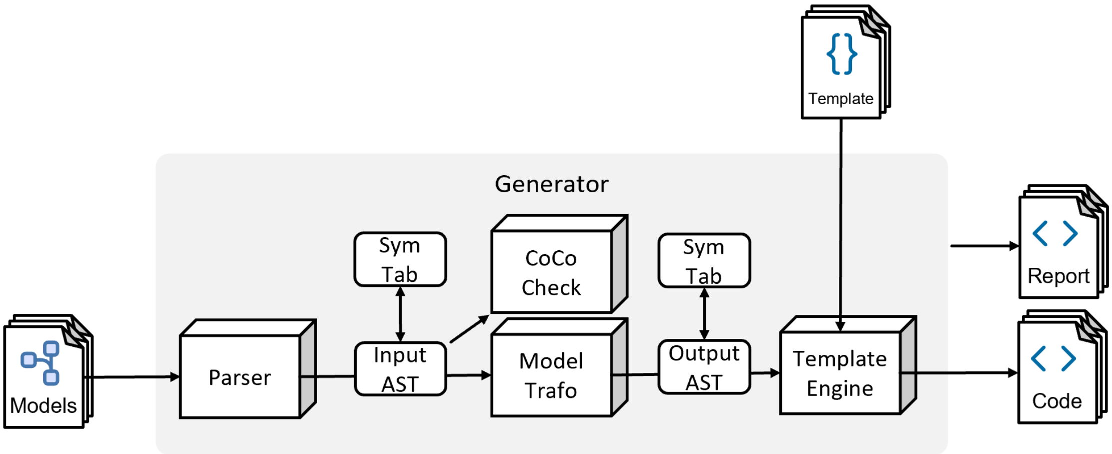


Figure 3.2: Typical architecture of a MontiCore based generator (Adapted from [HKR21]).

implemented by specific transformation languages as shown in [Wei12, Höl18, HRW15]. Finally, the generated template engine can be used to create source code in the desired target language using a set of templates. Next to the target code, MontiCore-based generators create a set of reports outlining information on the produced artifacts.

3.3.2 Symbol Table

In MontiCore, a symbol table plays a crucial role in the language infrastructure by storing information about symbols, used in the models created in a domain-specific language. Symbol tables in MontiCore are designed to support modularity, reusability, and extensibility, which align with its overall objectives as a DSL workbench [Völ11, HLMSN$^+$15b, HLMSN$^+$15a]. The MontiCore symbol table serves several important purposes, including symbol resolution, scope management, and model analysis. It helps resolve symbols within the models by managing the relationships between symbols and their definitions and handling symbol imports and inheritance. Additionally, MontiCore's symbol table assists in managing the scope of symbols, ensuring that symbols are accessible only within their appropriate context. Symbol tables also play a role in model analysis, facilitating tasks such as type-checking and consistency validation. They provide the necessary information about the symbols in the models, which is crucial to analyze the models and ensuring their correctness. Lastly, symbol tables are essential for code generation. They store information about the symbols and their associated attributes, which is necessary for generating correct and efficient code in the target programming language. [MSN17] defines a symbol table as follows:

Definition 6. ***Symbol Table***
„The ST is a data structure consisting of a scope graph with an associated collection of symbols at each scope. It maps names to essential information about model elements, represented as symbols. The ST allows to efficiently organize and find, among others, declarations, types, and implementation details associated with those model elements.“ (cf. [CBCR15])

The symbol table significantly enhances the efficiency of navigation between AST nodes within the given models. In addition, the symbol table encapsulates the core aspects of a language and its models. It embodies the model interfaces, which are an integral part of the language interface.

3.3.3 Templates

MontiCore has a template-based generator [Fow10]. It uses the FreeMarker[3] Template-Engine to generate its target code. The AST can be transformed over multiple transformations into the final source code. The final product itself can also be a generator as shown in Figure 3.2 (Input AST to Output AST) [HKR21]. Thus, MontiCore is also a generator that is used to generate generators. The template engine uses specific templates to specify how the target code is to be created. Each template consists of a static and a dynamic part, the latter being provided by the symbol table or by a visitor traversing the AST. [CFJ+16] describes a template as:

Definition 7. ***Template*** *[A template] is a raw piece of an artifact with explicitly marked expressions that are evaluated relative to a model and whose results are inserted to complete the artifact.*

3.3.4 Reports

MontiCore provides next to the target code a set of reports (Figure 3.2). The reports provide additional information summarizing meta-information on the generated source code. Reports can be used to help understand the operations of tools and artifacts that are generated with MontiCore. They differ from logs, as the reports do not provide step-by-step notes on the generation process but rather aggregate statistics on the finished product.

[3] https://freemarker.apache.org/

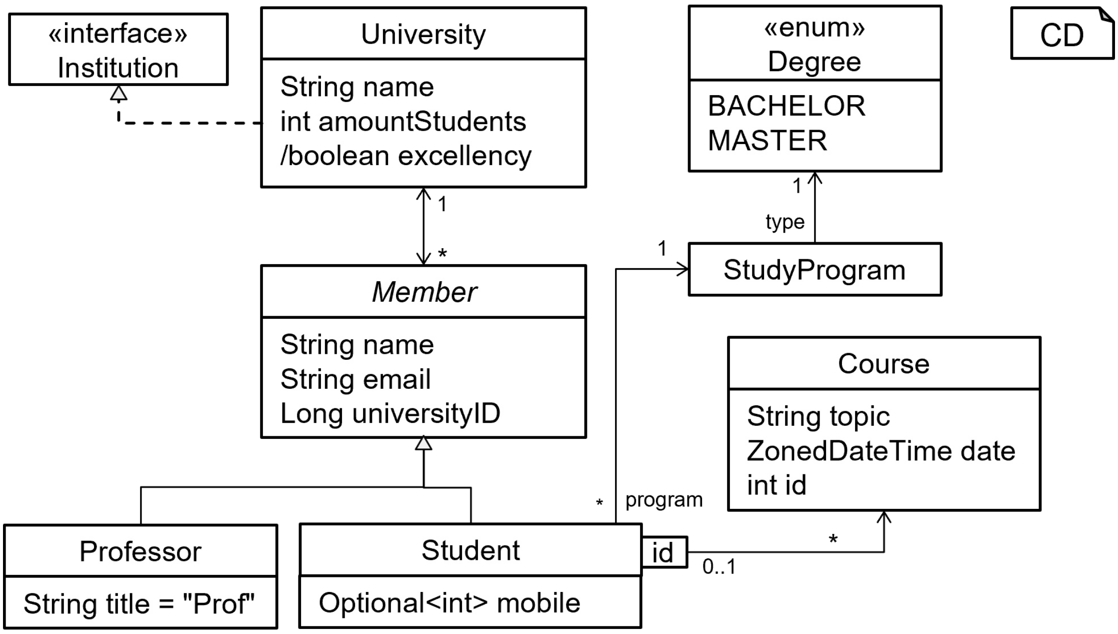


Figure 3.3: Class diagram describing a university

3.4 Class Diagram for Analysis

The DSL *Class Diagram for analysis (CD4A)* was developed by Roth [Rot17, MSNRR15a, MSNRR15b], to address the requirements of analysis models. The DSL was realized using MontiCore (*cf.* Section 3.3). In the following, we explain the CD4A language with an example. Figure 3.3 shows the relationship between a university and its members. The example describes a university (`University`) that has members (`Member`) who can be students (`Student`) or professors (`Professor`). Students are enrolled in a degree program (`StudyProgram`) and take courses (`Course`). Next, we build up this class diagram as a CD4A model step by step, highlighting the characteristics of the language.

3.4.1 Model Definition

Every CD4A model is defined in a single file with the same name as the model (such as in Listing 3.1). A CD4A model may begin with an optional `package` declaration (line 1) to group models into logical units and prevent conflicts. Imports, specified using the keyword `import` (line 3), allow the inclusion of external artifacts, including types not defined in the model.

```
1 package gem;

3 import java.time.ZonedDateTime;

5 classdiagram University {
6   // [...]
7 }
```

Listing 3.1: Basic structure of a CD4A model

We use the keyword `classdiagram` (Line 5) to define the main part of the model. Within brackets, we can define *classes*, *interfaces*, *enumerations*, and *associations* that belong to this model.

3.4.2 Classes, Interfaces and Enumerations

CD4A allows the definition of multiple classes within one model (*cf.* Listing 3.2). Each class is defined with the keyword `class` followed by the class name (lines 1-5). The keyword `abstract` (Line 1) as a modifier can be placed in front of a class to define abstract classes. A class can extend another class by adding the `extends` keyword and the targeted class after the class name (line 3). Similarly, an interface can be implemented using the keyword `implements` (line 5).

```
1 abstract class Member { ... }

3 class Student extends Member { ... }

5 class University implements Institution { ... }
```

Listing 3.2: Class definitions within a CD4A model. The classes can be `abstract` and extend each other. Interfaces can be implemented.

CD4A also allows for the definition of interfaces. They are defined with the `interface` keyword followed by its name. Listing 3.3 shows the definition of the interface `Institution` as shown in Figure 3.3.

```
interface Institution;
```

Listing 3.3: Interface definition within a CD4A model as shown in Figure 3.3

Interfaces may extend other interfaces by using the `extends` keyword as shown in Listing 3.4.

```
interface BasicShape;
interface Colored;
interface Polygon extends BasicShape;
interface ColoredPolygon extends Colored, Polygon;
```

Listing 3.4: An example using interfaces to define shapes: Interface extending other interfaces within a CD4A model

Next to classes and interfaces, enumerations can be defined in CD4A (*cf.* Listing 3.5). An enumeration is defined with the keyword `enum` followed by a name (line 1). Within its brackets, the enumeration defines a set of literals (lines 2-3). In CD4A an enumeration cannot inherit from classes, interfaces, or other enumerations; it also cannot have a modifier.

```
1 enum Degree {
2   BACHELOR,
3   MASTER;
4 }
```

Listing 3.5: Deinition of the `Degree` enumeration as shown in Figure 3.3

3.4.3 Attributes and Predefined Data Types

A class in a CD4A model may contain a set of attributes. Listing 3.6 shows the classes `University` (lines 1-5) and `Professor` (lines 7-9) and their attributes as shown in Figure 3.3. We can add the modifier `derived` or `/` in front of the type (Line 4) to define the types that themselves are computed based on other attributes. A derived attribute has a getter, but no setter or local variable declaration. In addition, default values can be defined. An attribute can be initialized with a primitive type such as `boolean` or `int`.

CD4A

```
class University {
  String name;
  int amountStundents;
  /boolean excellency;
}

class Professor extends Member {
  String title = "Prof";
}
```

Listing 3.6: Definitions of the classes `University` and `Professor` with their attributes as shown in Figure 3.3

Each attribute has a type, which must either be a primitive type, one defined in the model itself, or imported as an external data type. Listing 3.7 shows the definition of different types within a CD4A model. Types can be used in type constructors such as `List<...>` (line 2) or `Set<...>` (line 3) as well as `Optional<...>` (line 7). In addition to complex types, primitive types can also be used for generic collection types, for example `List<int>`.

CD4A

```
1 class MyCollections {
2   List<Long> longValues;
3   Set<String> stringValues;
4 }

6 class MyOptionals {
7   Optional<MyCollections> optionalValue;
8 }
```

Listing 3.7: Definitions several generic types within CD4A.

3.4.4 Associations

Associations describe the relationships between classes, interfaces, and enumerations. Listing 3.8 shows some of the associations depicted in Figure 3.3. In CD4A the association is defined with the keyword `association` followed by the end of the left association and an arrow that is directed (`<-`, `->`) or undirected (`<->`) or not specified (–) and finally the end of the right association. Cardinalities may be added to the left of the left association end and to the right of the right one. CD4A supports the following cardinalities: `[1]`, `[0..1]`, `[*]`, `[1..*]`. Further restrictions on cardinality have to be implemented by hand or modeled using OCL / P [BRW16]. Cardinalities are optional and are interpreted as underspecifications. Similarly, the roles of an association (*cf.* Line 2 (`program`)) are only necessary to prevent ambiguity; otherwise, the role can be derived from the type of opposite end of the association. The navigation direction is restricted in the cases of enumeration and external types, as it is not possible to navigate from those types.

```
association [1] University <-> Member [*];
association [*] Student (program) -> StudyProgram [1];
association [1] type StudyProgram -> Degree [1];
```

Listing 3.8: Definitions of associations as shown in Figure 3.3

In Figure 3.3 another type of association is shown: the qualified association. The access from one instance to a collection of instances via a qualifier. Listing 3.9 shows the qualified association from `Student` to `Course`. The qualifier is enclosed in double brackets: `[[id]]`. The double brackets denote that the qualifier (`id`) is a parameter of the target instance. In case an arbitrary parameter is used as a qualifier, single brackets are used.

```
association [0..1] Student [[id]] -> Course
```

Listing 3.9: Definitions of associations as shown in Figure 3.3

Further documentation for derived and ordered associations, as well as compositions, can be found in [Rot17].

3.4.5 Context Conditions

Context conditions are rules or constraints that govern the validity of language constructs within specific contexts or situations. Context conditions help to ensure that the elements of DSL are used correctly and in a semantically appropriate way, preventing errors or inconsistencies in the final model. For example, in the CD4A context, conditions prevent the use of an attribute that is neither imported nor defined. As there are several sets of context conditions defined for CD4A, we will only look at a few examples of context conditions of the DSL.

- **`CDAttributeUniqueInClass`** Within each class, an attribute must be unique. There must not be two attributes with the same name within a class.
- **`CDAttributeTypeExists`** The type of any declared attribute must be either an external data type that is qualified by an import statement, a primitive type or a type that has been declared in the model.
- **`CDInterfaceExtendsOnlyInterfaces`** Interfaces cannot extend classes. Interfaces may only extend other interfaces.

- **CDAssociationNameUnique** Within one class diagram, the name of an association must be unique.
- **RoleAndFieldNamesUnique** The role of an association must not have the same name as an attribute in one of the classes as either side of the association. This context condition also applies to implicit role names.

The context conditions in CD4A cover naming conventions such as capitalization of classes, the name of the diagram, or the uniqueness of names within the scopes. They also prevent circular inheritance or invalid extensions of enumerations or interfaces.

3.5 Tagging

In MDSE [Grö10, MNRV19] models are used as fundamental artifacts to focus on the application layer during development. The models are abstracted from technical configurations. In generative development approaches, models are often used as input for a generator [Cza02], to define how large parts of the application are generated automatically. To do so, the generator requires additional technical information, that is not part of the original input model and needs to be provided in another way. Greifenberg [GLRR15] initially developed the TaggingDSL to tackle this issue. The DSL can extend another modeling language and add additional information, by 'tagging' the symbols of the target model. This concept enables the separation of technical information from the input model. In the following, we take a look at the tagging concept, related work, and how it is used in the context of MontiGem.

3.5.1 Approach

The tagging concept for a DSL L is described in [GLRR15], it relies on three key artifacts: (1) the target model M_L that is to be extended by tagging, (2) the tagging schema $M_{L_{Schema}}$, a model that defines what elements of the target model can be tagged with which attributes, and finally (3) the tagging model $M_{L_{Tag}}$itself, defining which elements in the target model are tagged with which values. Figure 3.4 gives an overview of the languages and models involved in the tagging language. The lower part shows the languages defined as MontiCore grammars (MCG), and the upper part shows the corresponding MontiCore models (MCG). The diagram shows the relation between the language L_G and the tagging language derived from it. The language L_G^{Tag} and the tagschema L_G^{Schema} can be derived from the existing language L_G, by extending the grammar of the existing language L_G. Thus, any model M_{L_G} can be referenced from $M_{L_G^{Tag}}$, regardless of the language in which M_{L_G} is defined, as long as both models correspond to the same language L_G. Similarly L_G^{Schema} extends L_G allowing the model $M_{L_G^{Schema}}$ to reference elements of L_G.

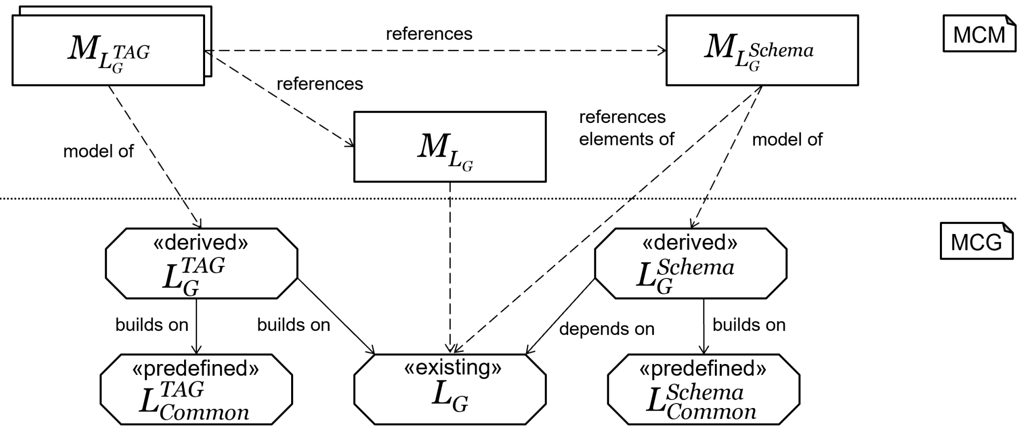


Figure 3.4: Relations between grammars and models of the tagging language (based on [GLRR15])

3.5.2 Common Tag Schema

```
1  grammar TagSchema extends Common {                                   MCG
2      TagSchema = "tagschema" Name
3          "{"
4              TagType*
5          "}"
6      ;
7      interface TagType;

9      Scope = "for"
10         (ScopeIdentifier ("," ScopeIdentifier)* | "*")
11     ;
12     SimpleTagType implements TagType =
13         ["private"]? "tagtype" Name Scope? ";"
14     ;
15     ValuedTagType implements TagType =
16         ["private"]? "tagtype" Name ":"
17         ("int"|"String"|"Boolean") Scope? ";"
18     ;
19     EnumeratedTagType implements TagType =
20         ["private"]? "tagtype" Name ":"
21         "[" String ("|" String)* "]" Scope? ";"
22     ;
23     ComplexTagType implements TagType =
24         ["private"]? "tagtype" Scope?
25         "{" Reference ("," Reference)* ";" "}"
26     ;
27     Reference = Name ":" ReferenceTyp ("?"|"+"|"*")? ;
28     ReferenceTyp = ("int"|"String"|"Boolean"|Name);
29 }
```

Listing 3.10: MontiCore grammar of for the common TagSchema [Loo17b], showing the four common TagTypes: `SimpleTagType`, `ValuedTagType`. `EnumeratedTagType` and `ComplexTagType`.

The tagschema allows the developer to define a type system for tags. Providing the domain expert with customized tools that ensure that all used tags are valid and allowing the generator developer to implement a generator that can process these predefined tag types. A tag schema is based on the grammar shown in Listing 3.10. A tagschema can consist of multiple definitions for TagTypes (Line 4). When defining a TagType, we distinguish in four different kinds of Tags: `SimpleTagType` (line 13) consists only of its name and is intended to flag only an element in the target model without any further information. `ValuedTagType` (Line 16) consists of a name and a value, allowing the developer to attach a value to a tagged element. Similarly to the ValuedTagType, `EnumeratedTagType` (Line 20) defines a name and a value, however, the TagType defines a set of which the provided value must be part. Finally, `ComplexTagType` (Line 27) defines a tag that can have an arbitrary number of subtags. These subtags can be primitive types such as `int`, `String` or `boolean`, but also other tag types already defined within the tagschema, allowing for unlimited depth in hierarchy and complexity.

3.5.3 Common Tags

```
MCG
1  grammar Tags extends Common{
2    TagModel =
3      "conforms" "to"
4        QualifiedName ("," QualifiedName)*";"
5      "tags" Name "for" targetModel:QualifiedName
6      "{" (contexts:Context | tags:TargetElement)* "}"
7    ;

9    Context = "within" ModelElementIdentifier "{"
10     (contexts:Context | tags:TargetElement)*
11     "}"
12   ;

14   interface ModelElementIdentifier;
15   DefaultIdent implements ModelElementIdentifier = QualifiedName;

17   interface Tag;
18   TargetElement = "tag" ModelElementIdentifier("," ModelElementIdentifier
        )* "with" Tag ("," Tag)* ";" ;

20   SimpleTag implements Tag = Name;
21   ValuedTag implements Tag = Name "=" String;
22   ComplexTag implements Tag = Name  "{" (Tag ("," Tag)* ";")? "}";
23 }
```

Listing 3.11: The MontiCore Grammar L^{TAG}_{Common} [Loo17b] defining common features of a tag.

The `TagModel` is based on the common tag grammar (Listing 3.11). It relates to at least one `TagSchema` (line 4) that has a `Name` and a `targetModel` (line 5). The grammar implements two key concepts: first the `Context` production, which allows the navigation into elements with the `within` keyword. Allowing the modeler to nest elements. Second, the `TargetElement` production, which defines how a specific element in `targetModel` can be tagged. The `TargetElement` consists of the keyword `tag`, one or more `ModelElementIdentifier` (lines 14) that has to be implemented in the sublanguage for the `targetModel` and finally one or more `Tags` that contain the actual tagged information.

Chapter 4

Automated Domain Modeling with Large Lanugage Models

As the definition of a model for a specific DSL can be a challenging task for an individual who is unfamiliar with software engineering, we will take a closer look at the usage of large language models in order to transform natural language into a model of a predefined DSL.

Contents

Yang and Sahraoui [YS22] explore the automatic creation of UML class diagrams using information extraction and model-driven methodologies for the classification of the natural language used as input. This method does not use LLMs and relies on rule-based transformation. Yang and Sahraoui describe their approach as having relatively low accuracy. Using LLMs we hope to improve upon these results.

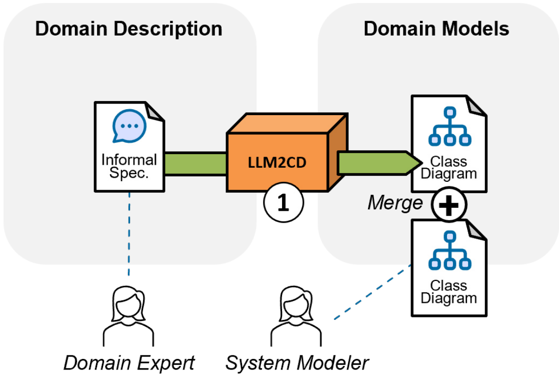


Figure 4.1: Except from Figure 2.4 (system architecture): First transformer of three. Transforming informal specifications into CD4A models.

As presented in Chapter 2 we use an LLM to transform the natural language input from domain expert into a data-structure model (cf. Figure 4.1). LLMs are based on stochastic models that in contrast to a deterministic algorithm can be imprecise and unpredictable. We have to treat LLMs as a black box that has a probability of yielding the desired result. Thus we have to investigate and measure the reliability of the used LLM to produce a model of a given DSL. In the following, we will take a closer look at large language models and their integration in our tool chain.

4.1 Large Language Models

A large language model is a type of artificial intelligence (AI) that is trained to understand, generate, and manipulate human language. It is typically built using deep learning techniques, particularly neural networks, and is trained on vast amounts of text data. The "large" in its name refers to the model's size in terms of the number of parameters it has, which can be in the range of billions or even trillions (*cf.* Figure 4.2, Table 4.1).

The language model learns the structure, grammar, and semantics of the language through exposure to the text data, allowing it to perform a wide range of language-related tasks. These tasks can include text generation, translation, summarization, question answering, and more. The text processed by the model is divided into small segments

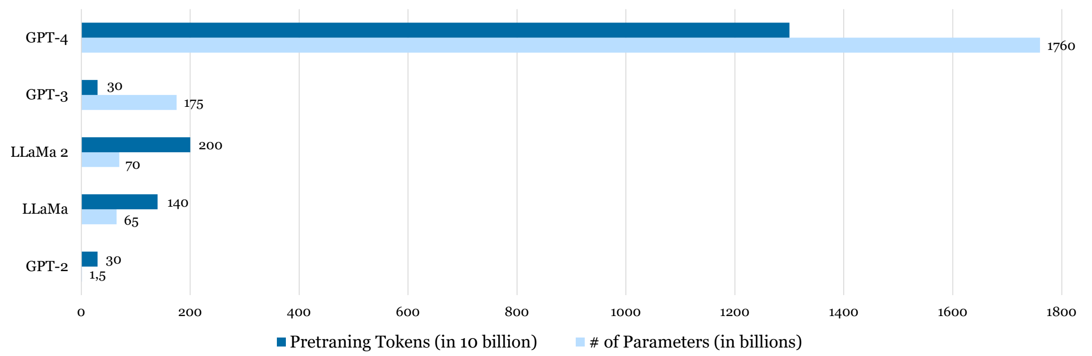


Figure 4.2: Size comparison of estimated parameter and token size of the currently largest language models available to the public.

Name	Release date	Produced by	Parameters	Open-sourced	Type
CodeBERT	February 2020	Microsoft	125M	YES	Encoder-decoder
InCoder	April 2022	Meta	6.7B, 1.3B	YES	Decoder-only
AlphaCode	February 2022	DeepMind	300M, 1B, 3B, 9B, and 41B	NO	Encoder-decoder
CodeX	August 2021	OpenAI	12B	YES	Encoder-decoder
Copilot	May 2022	GitHub and OpenAI	12B	YES	Decoder-only
CodeT5	October 2021	Salesforce Research	2B, 6B, 16B	YES	Encoder-only
PolyCoder	Oct 2022	Carnegie Mellon University	160M, 400M, 2.7B	YES	Decoder-decoder
CodeWhisperer	April 2023	Amazon	unknown	NO	unknown
CodeGen	March 2022	Salesforce Research	350M, 1B, 3B, 7B, 16B	YES	Decoder-only
StarCoder	May 2023	BigCode	15B	YES	Decoder-only
phi-l	June 2023	Microsoft	1.3B	NO	Decoder-only
Code Llama	August 2023	Meta	7B, 13B, 34B	YES	Decoder-only

Table 4.1: Existing Large Language Models [FGH+23]

called tokens. A token typically refers to a piece of text, such as a word or part of a word, that the model processes as a single unit of meaning or syntax. The LLM generates text by using an auto-regressive model: Predicts the upcoming token in a sequence based on the preceding tokens, thus generating coherent and contextually relevant text over longer passages. One of the key advantages of large language models is their ability to generalize within their training data. As the language model was trained on large amounts of data its capability to respond to text input is equally large. However, if prompted with input it was less trained on it will provide worse results this is referred to as *out of distribution generalization* (OOD). Within the context of machine learning OOD describes the ability of a model to perform well on a new, unseen dataset that differs in some ways from the data it was trained on. This concept is important because it relates to the robustness and versatility of the model when applied to real-world scenarios, where the data may not always match the conditions of the training set. Achieving OOD generalization is a significant challenge in machine learning because models tend to learn patterns specific to their training data. When faced with OOD data, these models might fail because they haven't learned the underlying, more general principles needed to handle the new

conditions. OOD is relevant to our use case as we provide the LLM with data set that are not yet publicly available (*e.g.,* models of new developed DSLs). Large language models do not yet have the capability to generalize upon data they were not trained on, as humans do. E.g. a model trained on the inventory restocking of a supermarket would be unable to predict inventory changes during the pandemic, as it was only trained on the regular shopping behavior of the customers The models we use within this work are trained on vast sets of data, making them versatile and powerful tools for many natural language processing (NLP) applications. However, they also have limitations, including the potential to generate biased or incorrect information, and they require careful handling to mitigate these risks. Context size and tokens are critical aspects of large language models that directly impact their performance and capabilities. The context size refers to the maximum number of tokens that the model can consider when generating a response. This is determined by the method used to train the model. A longer context size allows the model to maintain coherence over longer passages of text and to understand and refer back to information presented earlier in the text. However, increasing the context size also requires more computational resources, especially to train the model, making it a trade-off between performance and efficiency. Using a larger context on a model that was trained on a lower context size leads to a decline in response quality due to OOD.
The sentence "The quick brown fox jumps over the lazy dog." is tokenized using openAIs tokenizer[1] into 10 tokens as:

```
The quick brown fox jumps over the lazy dog.
```

Not only the input is tokenized, but all received and returned messages are used as context and separated into tokens. The following CD4A model (*cf.* Listing 4.1) is tokenized into 22 tokens as follows:

```
class University {
  String name;
  int amountStudents;
  /boolean excellency;
}
```

CD4A

Listing 4.1: Example of a simple class diagram defined in CD4A

```
class University {
  String name ;
  int amountStundents;
  /boolean excellency;
```

[1]https://platform.openai.com/tokenizer

```
}
```

Note that in contrast to tokenization of natural language, tokenization of source code might use up more tokens as it tends to separates words into multiple tokens and special characters such as brackets and semicolons require additional tokens as well. We introduce the following terms:

- **System message** The interaction with a language model is often formed as a dialog: A user provides input, and the language model responds to which the user can reply again. The system message can be used to initially provide guidance on how to use the API effectively, including formatting requests or handling output.

- **Priming** Priming refers to providing the model with a context or an example of the desired output before it generates a response. This can be done by including a specific prompt or set of instructions that 'prime' the model to respond in a certain way. For example, providing a detailed description of a style or tone can prime the model to generate text that matches these criteria. The effectiveness of priming depends on the complexity of the task, the specificity of the prime, and the underlying capabilities of the model. In language models, priming is a way to steer the generation process and can be crucial for applications where the context or style of the output is important.

- **Prompt** An Input provided to the language model that stimulates a response.

- **Parameters** refer to the elements of the model that are learned from training data. These parameters are essentially the internal settings or weights that the model adjusts during training. Each parameter influences how the model processes and generates text, contributing to its ability to understand language, context, and generate responses. The number of parameters in a model is a key factor in its capacity to handle complex language tasks, as more parameters generally allow for a more nuanced understanding of language.

- **Temperature** The temperature of a language model, is a hyper parameter that controls the randomness in the model's text generation. Hyper parameters, more broadly, are external configurations set before training a model. They guide the learning process but are not learned from data, unlike model parameters. Hyper parameters include settings like learning rate, batch size, and in the case of LLMs, temperature. They are crucial for optimizing model performance and require careful tuning. A low temperature leads to more predictable, conservative responses, while a high temperature results in more varied and creative outputs, but with an increased risk of irrelevant or nonsensical text.

- **Few-Shot learning** Few-shot learning is a concept in machine learning where a model is designed to learn information with a very small amount of data. Traditional machine learning models often require large datasets to learn from so that the patterns they recognize are robust and generalizable. However, in few-shot learning, the model must be able to make accurate predictions or understandings based on only a few examples. This approach is particularly important in situations where collecting large datasets is impractical or impossible, such as rare medical diagnoses, specific types of image recognition, or in the case of language models, understanding niche tasks or topics.

- **ChatGPT** ChatGPT is a platform optimized for chat-like conversation with a language model. ChatGPT itself relies on either GPT-3.5 or GPT-4. Table 4.3 shows the language models that were available at the time of writing. In the following GPT-3.5 refers to the gpt-3.5-turbo-0613 model and GPT-4 refers to the gpt-4-0613 model.

GPT-Modell	Technical Name	Max. Token	˜ DIN A4 Pages
GPT-4 (32k)	gpt-4-32k-0613	32,768	6.3
GPT-4 (8k)	gpt-4-0613	8,192	1.5
ChatGPT (16k)	gpt-3.5-16k-turbo-0613	16,384	3.0
ChatGPT (Standard 4k)	gpt-3.5-turbo-0613	4,096	0.8

Note: One DIN A4 standard page corresponds to 1,500 characters incl. spaces

Table 4.2: Different input lengths of selected models

4.2 Research Method

Due to the rapid advancements in the field of generative AI, and especially large language models, the development and evaluation of approaches in this area is challenging. An entire field of research is dedicated to the explanation of the behavior of these kinds of algorithm: Explainable artificial intelligence [DBH18, MWLN22]. As LLMs are statistical algorithms, we will rely mainly on empirical research methodologies to verify the validity of our approach.

Therefore, we rely on both *quantitative data analysis* and *qualitative data analysis* to evaluate each approach. In order to perform quantitative analysis, comparable and reproducible statistics are set up to reduce the influence of chance on the result. When evaluating results quantitatively, we primarily focus on the parsability of the given results. The prompts and configurations for the specific experiments can be found in the appendix in Section A.4. In addition to parsability, the quality of the produced artifacts is also evaluated. Currently, no generalizable method for evaluating AI-based models is known. Within this thesis, we rely on proven methods, such as grading schemas from exams, to measure model quality. These methods involve human supervision, and thus do not scale as well as the quantative analysis.

Currently, there is a need to standardize experimental setups involving LLMs Cámara, Burgueo, and Troya point out a suitable methodology in [CBT24].

4.3 Challenges

There are several challenges to using a large language model:

4.3.1 Using proprietary Language Models

Recent advances in LLM development have occurred primarily in the private sector. Due to the great progress made by OpenAI[2], especially in the context of GPT-4 [Ope23], there has been a great general interest in the development of further LLMs. At the time of this work, however, OpenAI has such a large lead with its GPT models (*cf.* Figure 4.2) that this work is primarily concerned with their models.

Cost of Operation: The training and operation of the newer generation of models require considerable computing power. Dedicated hardware is necessary to run an LLM with an equivalent size as GPT-3.5 or GPT-4. Thus OpenAI operates a paid Cloud-API. Since data is collected systematically and in large quantities in research, the costs incurred here can be an inhibiting factor.

Transparency: Since there are many direct commercial use cases for LLMs and competition for market leadership is fierce, OpenAI has no interest in disclosing algorithms, training data, or internal parameters. This within this work LLMs from OpenAI are treated as a 'black box' as almost no information of the inner workings of the provided models is known.

[2] `openai.com`

Reproducibility: The Cloud-API we can use gives us access to a specific model such as GPT-4 or GPT-3.5. However, we cannot use a specific version of that model. As OpenAI constantly optimizes and improves its models, we can not guarantee that an experiment will perform precisely the same if repeated after an update of the used model occurs. The usage of multiple different language models to generalize the validity of an experiment helps mitigate this challenge.

4.3.2 Limited Context Length

As mentioned in Section 4.1 the context of the language model is used to keep coherence over large input and is limited to a fixed number of tokens. In its default configuration, GPT-4 and similar transformers can process a context of up to 8.000 Tokens. There are language models that support up to 32.000 tokens (as of November 2023). However, these models were not publicly available at this time. The class diagram of a full-size application can easily reach the token limit. The CD4A-model [GHL+22] of the information system MaCoCo (*cf.* Chapter 8) consists of more than 100 classes and spans over 25.963 characters that would be broken down to 10.461 tokens. Thus this model could not be processed by the 8000-Token-Variant of GPT-4. There exist several approaches to tackle the challenge of a limited context size. Approaches such as AutoGPT [YYH23] can be used to break down and automate tasks. Different Large Language Models employ various methods to handle the challenge of processing long stretches of text. One such method is the Receptance Weighted Key Value (RWKV) [PAA+23], which builds upon Recurrent Neural Networks (RNNs) [Elm90]. RWKV creates a lasting state that is carried forward through each step of processing, akin to RNNs, but it operates at a more granular level, dealing with individual tokens. One of the key benefits of RWKV is its adaptability in handling texts of any length without a predetermined limit, and it manages to do so without increasing computational complexity. This contrasts with the current mainstream approach, transformers, which are widely used in LLMs but have their limitations. Despite their potential, RWKV systems have not yet received the same level of interest as transformers, resulting in a lack of large-scale models and comprehensive research to evaluate their performance over varying lengths of text. There is a hypothesis that RWKV models may experience a linear decline in performance as context lengthens, but this theory has yet to be thoroughly investigated and proven.

4.3.3 Hallucinations

Hallucinations refer to instances where the language model generates information or data that is not grounded in reality or is factually incorrect [MIT23]. This phenomenon occurs due to various reasons, such as the model's reliance on patterns it has learned during training rather than verifiable facts, or the model's attempt to fill in gaps in its knowledge with plausible-sounding but inaccurate information. The impact of hallucinations on the reliability of LLM output is significant. It undermines the trustworthiness of the information provided by these models, as users may not be able to easily distinguish between accurate information and hallucinations without cross-referencing other sources [MLG23]. A hallucination could occur as follows: The LLM is prompted: ”What is the main function of HTML in web development?” and it responds with ”HTML is primarily used to program the logic and functionalities of web applications,” this would be a hallucination. The correct answer is that HTML (HyperText Markup Language) is used for creating and structuring the content on the web, such as text, images, and links. It does not handle program logic, which is typically managed by languages like JavaScript. The response of LLM provided an answer that is plausible to those unfamiliar with web development, but it is factually incorrect. Hallucinations are the reason we need validation steps within our toolchain, to ensure the correctness of the produced output, as a simple plausibility check does not suffice.

4.3.4 Overfitting

The generation of MontiCore-based Models without specifically optimized prompting often leads to incorrect results both in GPT-3.5 and GPT-4 (*cf.* Zero-Shot Table 4.5). The language model often tends to produce models in PlantUML Syntax. As a result of OOD the language model has a tendency to reproduce data it was trained on rather than producing new data sets.

This can be attributed to PlantUML's significantly larger data foundation compared to MontiCore. Public sources have limited examples of MontiCore-based models, while PlantUML, a widely used open-source tool chain for UML diagrams, is present in various projects and organizations. Its straightforward text-based syntax and use in documentation, tutorials, and course materials provide a wealth of examples. The abundant examples could be utilized by OpenAI to train its language models.

Currently, MontiCore is primarily utilized in academic, research, and specialized industrial projects, limiting access to data compared to tools like PlantUML. As a result, over fitting may occur if the model is trained excessively on more common data, specifically PlantUML-Syntax, which may cause the model to struggle when generating less common data like MontiCore-based artifacts. Overfitting happens when a model is fine-tuned to a specific data set, leading to a loss of the ability to generalize to new and unknown data. This results in high accuracy when generating PlantUML artifacts, but low accuracy when generating MontiCore-based artifacts.

One solution to improve the accuracy of the model is by precise prompting. Clear and precise instructions can more effectively direct the model. For example, a prompt could specifically request a particular artifact, provide context or examples to help the model understand the requirements and increase the likelihood of generating a correct model. Effective prompting of a language model can be complicated and requires an understanding of the types of instruction needed to achieve accurate results. The developer of the toolchain must pre-identify effective prompts and integrate user input into useful prompts. Similar to programming, it is crucial to give precise instructions to language models that authentically reflect the user's intent. Unclear or ambiguous prompts can result in unpredictable or imprecise outcomes.

4.4 Transforming Informal Specifications to a Structured Model

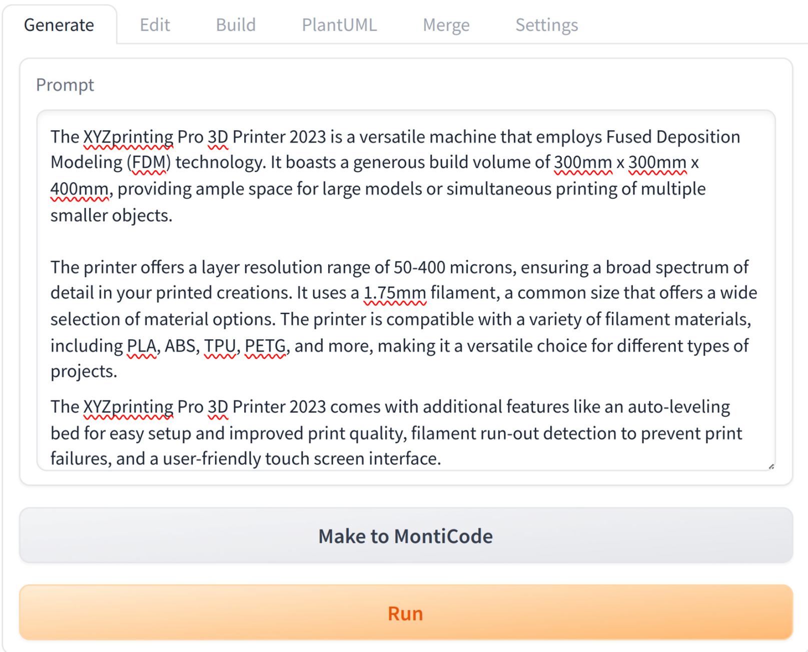


Figure 4.3: LLM2CD user interface for a domain expert used to provide informal specifications.

Although LLMs present many challenges, they also offer important functions that we can use for our transformation step. In the following, we consider a process chain that processes natural language as input, passes it to an LLM, post-processes the response, and finally outputs a CD4A model. Figure 4.4 presents the process of the transformer. A domain expert provides informal specifications as unstructured natural languages via a command line interface or for user convenience via an optimized web interface (*cf.* Figure 4.3). The domain expert can enter an informal set of specifications into a web client via a simple text field and start the transformation with one simple button (gray button in Figure 4.3). Any response from the tool chain is presented to the domain

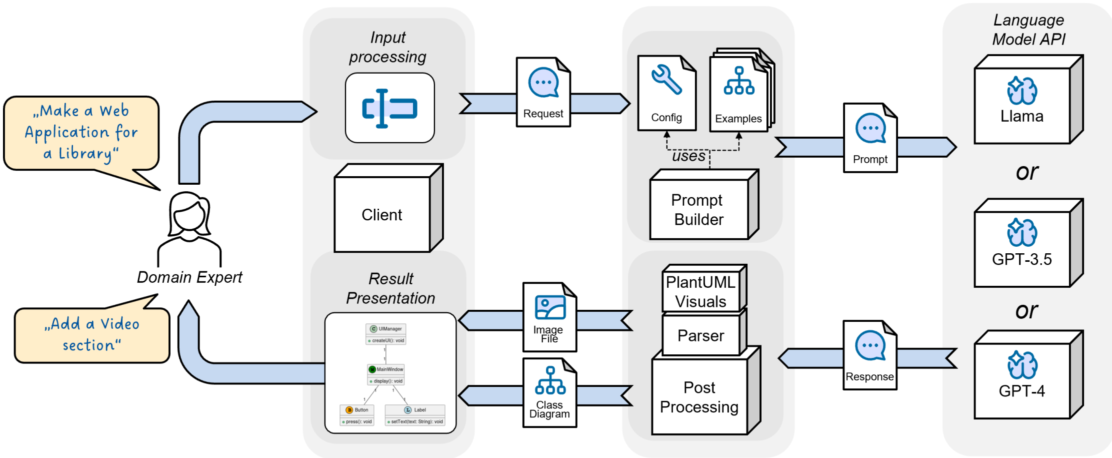


Figure 4.4: Transforming Natural language into CDs using priming, few-shot learning with provided examples, and post-processing. Based on [NMR24a]

expert in two forms: first, as the textual model and second as a visual representation as shown in Figure 4.5. A parser checks the validity of the model and indicates the result in the GUI as well providing the domain expert with direct feedback about the success or failure of the transformation. The domain expert can edit the textual variant and modify it before it is further processed. With the orange button below the input field, the domain expert can pass on the (edited) output to the next transformer, ultimately leading to the generation of a web application based on the produced class diagram. Let us take a closer look at the transformation from the domain expert input to a CD4A model: As shown in Figure 4.4, the input that the domain expert entered into the client is passed on to the *Prompt Builder*. We can use several methods to improve the chance of getting valid results from the language model. The prompt builder is provided with a

GPT-Modell	Technical Name	Max. Token	˜ DIN A4 Pages
GPT-4 (32k)	gpt-4-32k-0613	32,768	6.3
GPT-4 (8k)	gpt-4-0613	8,192	1.5
ChatGPT (16k)	gpt-3.5-16k-turbo-0613	16,384	3.0
ChatGPT (Standard 4k)	gpt-3.5-turbo-0613	4,096	0.8

Note: One DIN A4 standard page corresponds to 1,500 characters incl. spaces

Table 4.3: Different input lengths of selected models

configuration that contains access data for the used LLM-API as well as Priming data for the LLM. In addition, examples are provided in order to use the few-shot learning approach. Note that the choice of fitting examples has a high impact on the performance of this approach. The prompt builder assembles the prompt and passes it on to the configured LLM. The response is provided in a post-processing step. Here the model is extracted from the response message. Specific reoccurring errors are easier to fix in a post-processing step than to invest time and message space to train or prime the LLM in the initial prompt. Therefore, a list of checks and fixes can be included in this step. In addition, the CD4A model is transformed into a PlantUML model that can be easily transformed into a visual model. Both processed CD4A model and image are passed on to the domain expert for optional inspection.

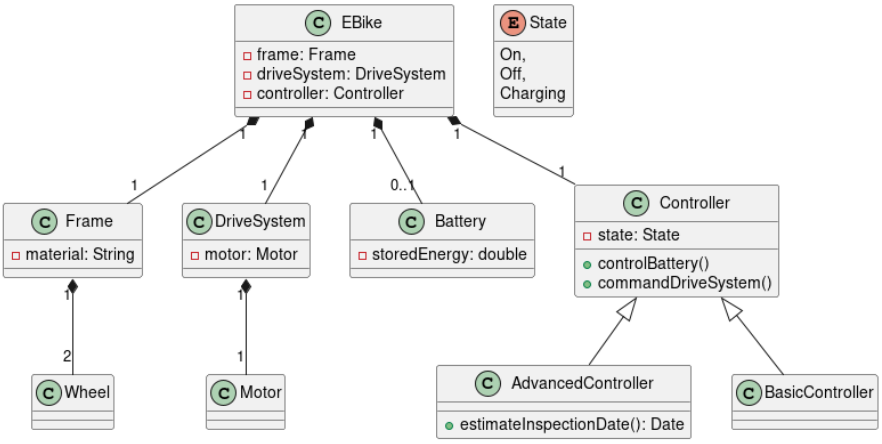


Figure 4.5: Visualization of a PlantUML Model extracted from a GTP-4 response. This model was generated as part of the 'E-Bike' use case (*cf.* Figure 4.8)

The toolchain utilizes the LLM to generate a valid CD4A model directly. This model is then applicable in subsequent transformation stages. While the language model exhibits superior performance in generating PlantUML models, we refrain from initially creating a PlantUML model to convert into CD4A. This is because the two languages are not entirely compatible—lacking 100% compatibility. Consequently, PlantUML serves merely as a visualization tool for the class diagram, and we continue our process with the CD4A model.

4.5 Modelling with Large Language Models

As we outsource modeling tasks to an LLM, we need to evaluate whether and how reliably LLMs can create models for informal requirements for a particular modeling language. We consider a synthesized model to be correct if both the syntax is valid and the semantics fit the original input. As there is a clear set of rules for the syntactical correctness of a model defined by the grammar of the used DSL, we can use a parser to validate syntax. MontiCore provides a parser for all DSLs defined with its tool chain.

Semantic correctness is harder to validate, as there is not yet a generic automated process comparable to a check via a parser. Therefore, we have to check each model by hand. For the evaluation, we will measure at what ratio a produced model is syntactically valid (in the following also referred to as *success rate*), and semantically correct. Several LLMs were evaluated and tasked to produce models for multiple DSLs. The following evaluations were performed

1. **Creating a new CD4A model:** Following the setup shown in Figure 4.4, we iteratively produce models based on informal specifications. We use established methods such as few-shot learning to provide the used LLMs with the necessary knowledge about the grammar of the target DSL.

2. **Creating a new PlantUML CD:** In order to generalize the results from the first evaluation, we analyze the performance of LLMs to create syntactically valid models on another DSL. Using the same method as before.

3. **Evaluating semantic correctness:** Having analyzed syntactic correctness, we provide ChatGPT with a task from an exam and assess the results based on a grading schema. Giving us a representative metric on semantic correctness and comparable data from student performance on previous exams.

4. **Passing parser feedback back to ChatGPT:** Up to this point, we have established to what rate individual LLMs can produce correct models for informal specifications. In the case of syntactical errors, the parser will state specifics of the encountered problem with the model. We evaluate in this section to what degree we can improve the success rate by iteratively providing ChatGPT with error messages of the parser.

5. **Adapting existing models:** Finally, assuming that it is feasible to use an LLM to create an initial model, we evaluate the success rate of modification of a textual model, based on a natural language input.

4.5.1 Creation of a CD4A model

To assess the capacity for generating syntactically accurate models, the identical query was executed multiple times (*cf.* Table 4.4). It is worth noting that these use cases were under-specified, causing the LLM to come up with its own solutions. A high degree of under specification is well suited to evaluate the capacity of a generative AI approach to create syntactically correct models [NMR24b, BDM+24]. However in order to evaluate semantic correctness we need to focus on use cases that specify a very small set of valid models. In the following we take a closer look at the syntax of the created models hence we can use very broad requirements that leave the LLM a high degree of freedom of interpretation. In an attempt to craft a CD for a *Ping Pong game* leveraging GPT-4 (as seen in the 4th row of Table 4.4), the resulting models demonstrated significant variability in terms of size (between 60 and 20 lines of code) and the nature of the game. We can use cosine similarity [RKA12] to determine how similar two texts are to each other. We can do so by converting each text into vectors. The most common way to do this is by using the bag-of-words model or TF-IDF (Term Frequency-Inverse Document Frequency) model. In a bag-of-words model, a text is represented as the bag of its words, disregarding grammar and even word order but keeping multiplicity. The TF-IDF model reflects how important a word is to a document in a collection or corpus. The similarity ranges from -1 to 1, whereas 1 indicates a very high similarity, and -1 indicates that the two texts are completely dissimilar. The computed average cosine similarity for all models if compared pairwise, stands at 0.52. An instance of the execution is presented in Figure 4.6.

User: *Please generate a Monticore CD4A model of a simple user interface.*

Predefined context: *A Monticore CD4A model starts with 'classdiagram' Name { where 'Name' should be the name of the model.* [...]

GPT-4: [...][a]

```
classdiagram PingPongGame {
  class Player {
    String name;
    int score;
  }
  class Ball {
    double speed;
    double direction;
  }
  class Game {
    Player player1;
    Player player2;
    Ball ball;
  }

  association Player -- Ball;
  association Player -- Game;
}
```

[a] Additional 'chat'-text is omitted to increase readability.

Figure 4.6: Interaction with ChatGPT: Creating a CD4A model

Model	Use Case	Valid	N	Rate
gpt-3.5-turbo-0613	Ping Pong game	15	20	75%
gpt-4-0613	Ping Pong game	99	100	99%
gpt-3.5-turbo-0613	Simple User Interfaces	76	100	76%
gpt-4-0613	Simple User Interfaces	97	100	97%
gpt-4-0613	Digital Twin of a Hydraulic Press	19	20	95%
gpt-4-0613	Lego Set 75192	19	20	95%

Table 4.4: Rate of syntactically valid models of different use cases evaluated over N iterations

To reduce the systematic error based on specific use cases, we ran further use cases (Table 4.4). GPT-4 has a high chance (97%) of providing a parsable model.
Both GPT-3.5 and GPT-4 exhibit a strong likelihood of producing a valid CD4A model. The LLM GPT-4 surpasses its predecessor in performance, owing to its more advanced language modeling capabilities [Ope23]. The success rate is consistent across different use cases, provided they share a similar level of complexity (For a more intricate use case, refer to Section 4.5.3). OpenAI has highlighted potential performance drops when the model encounters topics with limited or no prior training [Ope23]. While GPT-4 might reliably model a general product, it may falter and produce semantically inaccurate models for a specialized product having a distinct configuration. To evaluate this claim, we presented the LLMs with narrowly defined target domains such as:

User: *Please generate a model of Lego Set 75192.*

The prompt requires the LLM to generate a model for a very specific product, without giving additional information about the product in question (A Lego model of a Space Ship of the Star Wars franchise). The prompt was provided 20 times, in 19 cases the LLM provided a valid model. Only 4 described the specific Lego set corresponding to the corresponding ID (75192 : Millennium Falcon). The remaining models described generic Lego sets. We can conclude that an LLM also can produce models that correspond to very specific requirements or even unique identifiers.
We initially instructed the LLM to draft a CD4A model. [BMR+20, OKH+22] suggest that similar outcomes can be achieved by using examples instead of explicit instructions. As shown in Table 4.5, without any examples, both LLMs can fail to generate syntactically correct models. GPT-3.5 and GPT-4 are able to produce accurate results when provided with an example. Interestingly, GPT-3.5 performs worse when presented with more examples, whereas GPT-4's performance is enhanced with additional models. We assume that this is due to over-adaption to the examples and out-of-distribution generalization as well as the reduced context size of the smaller LLMs.

Language Model	Kind of Context	Valid	N	Rate	Similarity
gpt-3.5-turbo-0613	Zero-Shot	0	100	0%	0.08
gpt-3.5-turbo-0613	One-Shot	51	100	51%	0.20
gpt-3.5-turbo-0613	Few-Shot	26	100	26%	0.22
gpt-4-0613	Zero-Shot	0	20	0%	0.12
gpt-4-0613	One-Shot	73	100	73%	0.25
gpt-4-0613	Few-Shot	88	100	88%	0.32

Table 4.5: Rate of syntactically valid models generated for the same task (Creating a CD4A model) using different contexts, evaluated over N iterations. Similarity is based on the pairwise average cosine similarity of all generated models.

The following prompt is used to apply the few-shot learning approach to a LLM:

````
[
    {
        "role": "system",
        "content": "You create cd4a class diagrams based on examples"
    },
    {
        "role": "user",
        "content": "Create a cd4a file of a simple user interface"
    },
    {
        "role": "user",
        "content": "Here are example of  cd4a files do not use PlantUML syntax: example1:{example1} , example2:{example2}, example3:{example3}"
    },
    {
        "role": "assistant",
        "content": "I will always create the cd4a file and not ask additional questions. I will surround the diagram code with ```. start the code the line after ``` "
    }
]
````

Figure 4.7: Prompts applying few-shot learning to produce a cd4a class diagram

The LLM is very unlikely to produce the correct syntax if only provided with the task to transform the specification to a specific DSLs, assuming that the DSL is not widely used and thus the LLM is less likely to be trained on that DSL. We can teach the LLM by either giving examples of the DSL (Few shot learning) or by providing it with a few instructions on how to adhere to the syntax. [WTB+22] describes this effect as *emergent* *abilities*. The model was not specifically trained to solve these tasks and this effect increases with the size of the language model.

4.5.2 Creating a PlantUML CD

We achieved a success rate of up to 99% for CD4A models using GPT-4, as documented in Table 4.4. The DSL in question is tailored for developers and bears a syntax strongly reminiscent of Java. Given that the language model has also been trained on Java code, it is plausible that this facilitates generating code with a Java-like syntax, leading to an impressive success rate.
To further probe this capability, we explore its performance on a different DSL: PlantUML [Arn23]. Unlike MontiCore based CD4A, the PlantUML language encompasses various diagram formats, including CDs, state charts, activity diagrams, BPMN, and component diagrams. Consistent with our methodology for evaluating CD4A models, we employed a parser to ascertain the syntactic accuracy of the generated models.

Model	Use Case	Valid	N	Rate	Similarity
gpt-4-0613	Ping Pong game	20	20	100%	0.33
gpt-4-0613	Simple User Interface	18	20	90%	0.24
gpt-4-0613	Exam Task	18	20	90%	0.63

Table 4.6: Success rate of different use cases evaluated of N iterations. The Cosine Similarity of models increases with more restrictive use-case descriptions. 'Ping Pong game' and 'Simple User Interface' are used as task descriptions 'as-is'. 'Exam Task' is presented in detail in the Appendix at Section A.4.4.

Table 4.6, shows that the variation in data structure describing DSLs has a minimal impact on the approach's success rate. The achieved success rates (90%-100%) align closely with those observed for CD4A.

4.5.3 Evaluating Semantic Correctness

Assessing semantic correctness remains a challenge (*cf.* [VAT$^+$20]) and is frequently debated. To provide a practical measure, we provide the system with an examination task

User: *Generate a class diagram of an EBike according to these specifications:*
The EBike is composed of a frame (made out of steel), a drive system, and a controller. Two wheels are inserted into each frame. The drive system is composed of a motor. Each EBike can be connected to a removable battery. The battery has a stored energy measured in Watt-hours (Wh). The controller can be in one of three states: On, Off, and Charging. It also controls the battery, if one is connected, and commands the drive system. The company plans two different variants of the controller, a basic controller, and an advanced controller. The advanced controller should be able to estimate the next Date the bike should be inspected for maintenance.

GPT-4:

```
@startuml

  class EBike {
      frame : Frame
      driveSystem : DriveSystem
      controller : Controller
      battery : Battery
  }

[...]

@enduml
```

For complete model *cf.* Figure 4.9

Figure 4.8: Exam task, describing several specifications for an EBike.

By utilizing the exam's grading criteria, we compare the LLMs output against students' performances who undertook the same exam. This specific task is scored out of a possible 13.5 points, evaluating not only the diagram's completeness but also key elements like compositions, inheritance, cardinalities of associations, and common design pitfalls.
We evaluated 20 models generated by ChatGPT, with an average score of 7.5 out of 13.5 points. Scores ranged from a low of 5 points to a high of 10.5. Models that scored less than 50% of the available points 'failed' the test. Thus three of the generated models were deemed unsuccessful. In comparison, a set of 40 students who took the exam achieved an average score of 10.0 points. While the task is regarded as an entry-level task for the students—with the majority scoring above 50%—it underscores that our method can yield models that closely align with specific requirements.

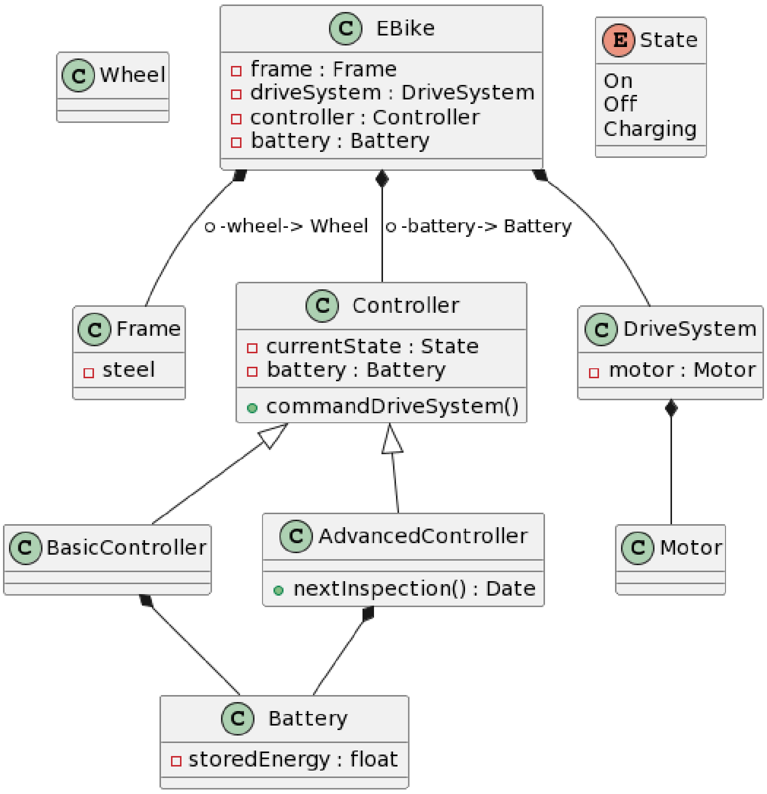


Figure 4.9: Visualization of the Textual Model extracted from a GTP-4 response. This model was graded with 8.5/13.5 Points according to the grading schema of the exam. (*cf.* Figure 4.8)

4.5.4 Iteratively correcting models using an LLM

Up to this point, we know that LLMs have a certain chance to produce a correct model. Next, we evaluate methods to fix incorrect model that were produced by a language model:

Correcting a Model by Informal Request

The LLM can be used to correct an existing model. We tasked it with extending a provided CD without specifying the employed DSL (*cf.* Figure 4.10). In the following step, another adaptation was issued. In both instances, GPT-4 generated valid models. Yet, its method of 'optimization' was to merely comment out the problematic code (as seen in the second response's fourth line), an approach often viewed as poor practice.

User: *Add to the following class diagram a new* `class` *'*`dog`*':*

```
classdiagram Flensburg {
  class Human { int age; }
  class Person extends Human {
    String nickname;
  }
}
```

GPT-4: [...]

```
classdiagram
    class Animal {
        int age;
    }
    class Human extends Animal {
        int age;
    }
    class Person extends Human {
        String nickname;
   }
   class Dog extends Animal {
       String name;
       String breed;
       [...]
   }
}
```

User: *Both* `Animal` *and* `Human` *have the attribute* `age` *and* `Human` *inherits from* `Animal` *please optimize the model.*

GPT-4: [...][a]

```
class Human extends Animal {
// removed because inherited from Animal
// int age;
}
[...]
```

[a] Note that GPT-4 provided the complete model, we reduced the model to the relevant class to improve readability.

Figure 4.10: Using GPT-4 to adapt models

Using Parser-Feedback to Correct Syntactically Incorrect Model

We have shown that an LLM can be used to correct models based on informal feedback. In case we provide erroneous models to a model-driven software development framework, we will receive error messages. These messages can be used to task an LLM to correct the model it provided. Thus, we evaluate next if we can employ the parsers feedback to rectify syntactically invalid models. We use a parser to evaluate if a model is correct. In case it is not, a new prompt is assembled that contains both the input model, the corresponding error message of the parser, and instructions to the LLMs on how to handle the input. The result from the LLM is sent to a parser again, in case the model is still syntactically wrong, we repeat the process with the new error message from the parser, otherwise, the valid model is passed on to the framework to be processed further. The resulting prompt is structured as follows:

Predefined context: *You will get a MontiCore CD4A model then I will pass you parser errors and you will correct the model with the help of those errors.*

User: *Here is the model:* [...][a]

[a] In order to maintain readability, we omit the CD4A model

GPT-4: *Please provide me the Syntax Errors*

User: *Here are the syntax Errors:* [...][b]

[b] In order to maintain readability, we omit the specific parser Errors. An example can be found below.

GPT-4: *I will now provide the code that I have corrected with the help of the syntax errors*

We use the following setup: the domain expert defines a use case. The informal use case description is sent together with the initial context to ChatGPT. The CD4A model is extracted from the response and provided to the parser. The parser either passes on the model to a generator or returns an Error. In case there is an error, it is paired with the context shown above and sent to ChatGPT together with the invalid model, in order to fix any problematic lines. Out of more than 10 attempts to correct an invalid model, none succeeded within our automated loop, with each running up to a maximum of 100 iterations between parser and LLM.
Let us take a look at the following example: the LLM provides the user with a CD4A model with invalid syntax: Instead of `classdiagram`, it places `class` at the beginning of the model:

GPT-4: [...]

```
class Pong {
  class Player {
    private int score;
    [...]
    }
}
```

We return the following parser error together with the context mentioned above back to GPT-4:

Parser: *Here are the syntax Errors:*
```
[ERROR] 0.cd:<2,0>:
extraneous input 'class' expecting '«', 'protected',
'readonly', 'package', 'static', 'final', '?',
'private', 'import', 'public', 'abstract', Name in
rule stack:  [CDCompilationUnit]
```

In this example GPT-4 has difficulties correcting the mistake, as the error message itself does not contain the correct hint for the solution and is misleading. Thus the resulting model produced by the LLM will not parse. The system is stuck in a loop.

GPT-4: [...]

```
CDCompilationUnit {
    class Pong {
      class Player {
        private int score;
        [...]
      }
    }
```

The non-determinism of the algorithm could help here to break free from this loop, but after an average of 4.5 runs, the loop ”degenerates” as ChatGPT insists that there are no more syntax errors. ChatGPT can get stuck in a loop stating the following refusal to work:

GPT-4: *I am sorry, but it seems like you have not provided the complete code or the syntax error you have provided is not related to the code you gave me. [...] Apology as there was no syntax error in your given MontiCore model " Pong ". Can you give me a valid MontiCore model with Syntax errors so that we can move forward?*

Figure 4.11: GPT is unable to further debug a model.

Example of a degenerated model, containing multiple lines of comments and very few LoC, while still being invalid.

GPT-4:

```
class Paddle {
  private int paddleSize;
  // Updated method signature to be
  // grammatically correct (added name)
  public void setPaddleSize(int size) {
  // Update assignment statement to use
  // 'this.' to refer to variable
  this.paddleSize = size;
 }
```

LLM	Use Case	Runs	Valid
gpt-3.5-turbo-0613	Ping Pong (missing brackets *cf.* Listing A.17)	9	0
gpt-3.5-turbo-0613	Ping Pong (missing brackets *cf.* Listing A.17)	2	0
gpt-3.5-turbo-0613	Ping Pong (missing keyword *cf.* Listing A.18)	20	0
gpt-3.5-turbo-0613	Ping Pong (missing keyword *cf.* Listing A.18)	17	0
gpt-3.5-turbo-0613	Ping Pong (erroneous association *cf.* Listing A.19)	2	0
gpt-3.5-turbo-0613	Ping Pong (erroneous association *cf.* Listing A.19)	8	0
gpt-3.5-turbo-0613	Ping Pong (namespace violation *cf.* Listing A.20)	7	0
gpt-3.5-turbo-0613	Ping Pong (namespace violation *cf.* Listing A.20)	5	0

Table 4.7: Returning parser feedback to GPT did not lead to valid models. The process was iterated until either a valid model was produced or the system stuck in a loop losing the model (*cf.* Figure 4.11). There is a high tendency that further errors were introduced into a invalid model.

With the current configuration, it is unfeasible to use the parser feedback to optimize faulty models. Improvement of the error messages by the parser would improve the performance of the LLM.

4.5.5 Adapting Existing Models

We presented methods above to create models using LLMs. To evaluate the usage of LLMs in modeling, we need to analyze to what degree LLMs can be used to modify already existing models.
Our method is highly probable to produce diverse, suitable model variants tailored to specific use cases, making it ideal for prototyping or initializing a software project. However initial versions of a model often require adjustment and refinement. Given this, it is crucial to assess whether parts of a model can be extended or adjusted based on domain expert preferences and upcoming ideas.
To evaluate this, we utilize a CD from one of our research endeavors comprising roughly 30 classes (for an excerpt of the model, refer to [MNRV19], the complete model is shown in Listing A.2). We then direct our tool to introduce a particular class and establish a new context to guide ChatGPT in augmenting the model with accurate syntax.

> **User:** *Add a user class and an association between the user class and* `Person` *to this model:* [...]

Figure 4.12: User prompt to ChatGPT

In 20 iterations, 19 produced correct PlantUML models, each incorporating a user class. With CD4A, all 20 models were successful. A noteworthy observation was ChatGPT's adaptation of model-specific attributes. For instance, within our model, the atypical `ZonedDateTime` type was employed for time-centric attributes. When ChatGPT designed attributes related to time, it mirrored this type (e.g. for an attribute `dateOfBirth`). A similar trend was observed in commenting styles. The provided CD is segmented into sections demarcated by large comment blocks (e.g., `/*******` ...); ChatGPT not only replicated this style but also tailored it to its modifications (*cf.* Figure 4.13).

GPT-4: [...]

```
[...]
  /************************************
   * New class and association
  ************************************/
  class User {
    String name;
    String email;
    String role;
  }

  Association User <-> Person [*];
[...]
```

Figure 4.13: Response from ChatGPT

Repeatedly iterating over the same model with commentary can pose challenges. ChatGPT occasionally omits elements by simply commenting them out, leading to compromised models akin to those detailed in Section 4.5.4. Fine-tuning the context to prevent ChatGPT from inserting comments might alleviate this issue.

Conclusion

LLMs can be used to perform modeling tasks and have a high chance of producing correct models for data structures. We can also use LLMs to edit and correct models iteratively based on informal specifications. We are unable to automatically use parser feedback to fix syntactically incorrect models.

4.6 Limitations

Although Large Language Models possess exceptional capabilities that pave the way for technological advancement, it is crucial to identify their limitations to comprehend and responsibly harness their potential.

Issues Stemming from Non-Determinism

Most challenges associated with non-determinism are rooted in an inadequately specified task for ChatGPT. Configuring the API to persistently use the same context for every request and limiting ChatGPT to a designated target-DSL has led to consistent and favorable outcomes. The influence of nondeterminism can be measured by the average variance among generated models via cosine similarity. Restricting the use-case description, as demonstrated in Table 4.6, can diminish this variance. A well-defined task tends to yield more homogeneous models. Note that a consistent temperature setting of 0.8 was maintained for the API across all tests. The temperature of the language model can be set to 0.0. Doing so would render systematic evaluation useless as all runs with the same prompt return identical results. As the language model used is updated over time, the model would still yield varying results. Keeping the unpredictability and to a certain degree the non-determinism (due to language model updates) but losing the ability to do a statistical evaluation. As we propose iterative usage of LLMs we have to take a closer look at to what degree little changes to the prompt result in large changes in the produced model. In order to evaluate this effect, several variations of the same prompt were given to a GPT-3.5 model. Then the mean similarity of the resulting models was computed

> Create a CD4A file for the homework submission system of a school. The system should include features for student submissions, teacher reviews, and grading. It should support multiple file formats and ensure that submissions are timestamped. Include sections for student registration, assignment creation by teachers, submission by students, review by teachers, and feedback provision.

Figure 4.14: Baseline promt

The following prompt is derived from the baseline prompt. However, it describes the same use case and contains many synonyms (underlined in the prompt).

> Develop a CD4A document for the <u>homework management</u> system of a school. The system should <u>encompass functionalities</u> for student submissions, <u>instructor evaluations</u>, and grading. It <u>must accommodate various</u> file types and <u>guarantee</u> that submissions are time-stamped. <u>Incorporate</u> sections for student <u>enrollment</u>, assignment formulation by <u>instructors</u>, submissions by students, <u>evaluations</u> by instructors, and the provision of feedback.

Figure 4.15: Promt with synonyms (underlined passages in prompt)

Prompts that describe almost identical use cases yield very similar class diagrams. On average, class diagrams ($n = 5$) that were based on semantically similar prompts have a similarity of 0.8 showing an almost identical set of attributes and tend to diverge stronger when defining associations.
We can conclude that wording or spelling has little effect on the created model, as long as the semantics of the prompt remain identical or at least very similar. However, we cannot guarantee that the LLM will always produce a predictable model.

Issues Pertaining to Accuracy

As highlighted earlier, one inherent risk with ChatGPT is its potential to produce inaccurate results. In our experiments, we only verified the syntactical integrity of the model, delegating semantic validity checks to domain experts. With the aid of a parser, we can confirm the syntactical soundness of each response. Concerning semantic accuracy, our tests indicate a high probability of ChatGPT offering suitable results, though absolute correctness for every run cannot be assured.

Challenges with Unfamiliar DSLs and Scalability

We assessed this method using two DSLs related to a standard UMLP Diagram: the class diagram. Some positive outcomes might be attributed to ChatGPT's familiarity with UMLP concepts. It is plausible that designing models for alternative DSLs, such as UI-description languages, might pose more challenges. Nonetheless, we observed ChatGPT's capability to adjust based on provided input (Section 4.5.5), which can be enhanced with an appropriate context. Further exploration is warranted.

Employing Multiple Sequential Generative Techniques

Existing LLM-based solutions like CoPilot [BJP22] and Codex [SVD21] generate source code from natural language descriptions. However, this is not an alternative to our presented method. Using a model-driven approach has the benefit of defining large structures while only requiring only a few artifacts. Model leverage higher level abstractions can be used to deterministically generate large quantities of source code. However, direct code generation with LLMs, remains confined to relatively small code segments and is best suited for smaller tasks, such as in educational settings [FADB$^+$22].

Prompt Injection

Prompt injection is a technique, where a user intentionally crafts or modifies the input prompt to influence or manipulate the model's response. This can be done for various purposes: *Guiding the Response:* By adding specific instructions or context to the prompt, users can guide the model to respond in a certain way or focus on particular aspects of a topic. *Changing Behavior:* Injecting certain phrases or keywords can alter the model's tone, style, or even the type of content it generates, like asking it to answer in the style of a specific genre or character. *Bypassing Restrictions:* Sometimes, users attempt to use prompt injection to circumvent built-in content filters or guidelines, though robust models are often designed to recognize and resist such attempts. LLM4CD is susceptible to prompt injection as well. However, as we produce a model that is validated by a parser, the options to misuse the language model are very limited.

Inherent Limitations

While our intention is to minimize the dependency of the domain expert, we cannot entirely rule out potential misuse. As we rely on the domain expert's written specifications, we cannot validate user input, thus ensuring accurate output becomes challenging. For instance, assuming the user does not dictate syntax, misaligned input, like requesting a different DSL than what the tool is set up for, will likely yield suboptimal results. We aim to refine this tool by introducing features for easier utilization. Furthermore, we're bound by the LLM's current context capacity. ChatGPT-API, in its present form, can only accommodate up to 8,000 tokens. The class diagram in Section 4.5.5 spanned 30 classes, translating to 1,647 tokens. A more extensive diagram such as the MaCoCo class diagram (*cf.* Listing A.1) with 100 classes, equating to 12,031 tokens, would exceed this limit. Anticipated updates promise a 32,000-token capacity, allowing for more expansive models. Lastly, ChatGPT's knowledge is confined to its last training cut-off in 2021. Information post this period, or not publicly accessible, remains outside its scope. While future updates to the dataset are conceivable, we currently operate with the 2021 dataset. OpenAI has hinted at plugins to enable ChatGPT to fetch real-time internet data.

4.7 Using LLMs for MDSE

Up to this point, we only focused on the usage of LLMs to synthesize and edit class diagrams. However, the assumption is reasonable that if a formalization of informal requirements in class diagrams is possible, a similar formalization in other DSLs may also be possible. In the following, we will take a look at a number of evaluations of this approach for other DSLs.

4.7.1 MontiCore Feature Diagrams

The DSL is based on UML feature diagrams and follows its key principles [CE00, 50101]. We provided GPT-3.5 and GPT-4 with the needed examples for the few-shot approach (*cf.* Listing A.13, Listing A.14, Listing A.15, Listing A.16) and requested a feature diagram for multiple use cases (results see Table 4.8).
The prompt was set up as follows:
GPT-4 performs better in both 1-Shot and 2-Shot, than GPT-3.5. The majority (more than 80%) of the models produced by GPT-4 are valid. Figure 4.17 gives an example of one produced feature diagram for the hydraulic press use case

4.7.2 MontiCore Sequence Diagrams

Similar to the feature diagrams presented above, the performance of the approach can be evaluated for the sequence diagram DSL [Obj17]. In this case the LLM was tasked

```
[
    {
        "role": "user",
        "content": "Create a MontiCore feature diagram for a Hydraulic Press"
    },
    {
        "role": "user",
        "content": "Create a sequence diagram using MontiCore syntax to
        ↪ describe a typical bar visit"
    }
]
```

Figure 4.16: Prompts producing a feature diagram

Test Name	Model	Models containing Syntax Errors (N)[1]	Test runs [2]
Hydraulic Press 2-Shot	GPT-4	10.00 % (3)	30
Hydraulic Press 1-Shot	GPT-4	14.29 % (3)	21
Ping Pong 1-Shot	GPT-4	12.33 % (3)	25
Falcon 1-Shot	GPT-4	18.75 % (3)	16
Ping Pong 2-Shot	GPT-3.5	40.00 % (14)	35
Falcon 2-Shot	GPT-3.5	42.86 % (12)	28
Hydraulic Press 1-Shot	GPT-3.5	45.83 % (22)	48

1: Absolute number on syntactically incorrect models. 2: Absolute number of created models.

Table 4.8: Percentage of erroneous feature diagrams, produced with our approach with both 1-Shot and 2-Shot learning. A feature diagram is erroneous if the parser detects a syntactical error. We do not cover semantic errors in this table.GPT-3.5 denotes gpt-3.5-turbo-0613, GPT4 denotes gpt-4-0613.

to produce a sequence diagram of the review process of a conference paper:
The prompt was set up as shown in Figure 4.18. The models used as examples can be found in the Appendix at Listing A.9, Listing A.10, Listing A.11, Listing A.12.

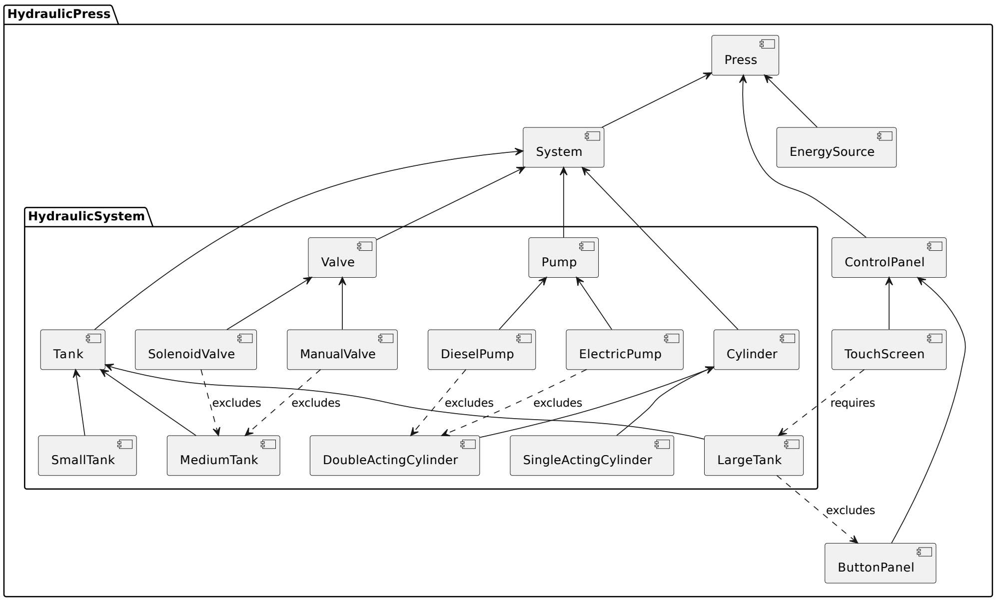


Figure 4.17: PlantUML visualization of the Hydraulic Press use case (Rendered with `https://www.planttext.com/`).

```
[
    {
        "role": "system",
        "content": "You create sd files based on examples"
    },
    {
        "role": "user",
        "content": "Create a sd file of the submission and review process of
        ↪  the Workshop Modeling in the Age of Large Language Models
        ↪  (LLM4Modeling) conference."
    },
    {
        "role": "user",
        "content": "Here are example of  sd files do not use arrays or lists
        ↪  and keep it simple: example1:{example1} , example2:{example2},
        ↪  example3:{example3}"
    },
    {
        "role": "assistant",
        "content": "I will always create the sd file and not ask additional
        ↪  questions. I will surround the diagram code with ```. start the
        ↪  code the line after ``` "
    }
]
```

Figure 4.18: Prompts producing a MontiCore sequence diagram

Test Name	Model	Models containing Syntax Errors (N)[1]	Test runs [2]
Review 1-Shot (PUML Mitigation)	GPT-4	32.5 % (13)	40
Review 2-Shot	GPT-4	30.0 % (12)	40
Review 3-Shot	GPT-4	40.0 % (16)	40
Review 3-Shot	GPT-3.5	50.0 % (25)	50
Review 2-Shot	GPT-3.5	60.0 % (12)	20
Review 1-Shot	GPT-4	67.23 % (80)	119
Review 1-Shot	GPT-3.5	78.33 % (47)	60
Review 1-Shot (PUML Mitigation)	GPT-3.5	80.0 % (48)	60
Review 0-Shot	GPT-3.5	100 % (34)	34

1: Absolute number on syntactically incorrect models. 2: Absolute number of created models.

Table 4.9: Percentage of invalid sequence diagrams, produced with our approach with both 1-Shot and 2-Shot learning. As the creation MontiCore sequence diagrams is especially susceptible to overtraining (*cf.* Section 4.3.4), additional fine-tuning was used to avoid PlantUML syntax.
GPT-3.5 denotes gpt-3.5-turbo-0613, GPT4 denotes gpt-4-0613.

In comparison to feature diagrams and class diagrams, sequence diagrams are harder to produce with an LLM (*cf.* Table 4.9). Similar to previous evaluations, GPT-4 performs significantly better than GPT-3.5. Both LLMs have a strong tendency to produce PlantUML syntax instead of the specified MontiCore sequence diagrams. Therefore the configuration was updated in order to mitigate this problem (*cf.* Section 4.3.4). The results improved accordingly.

4.7.3 MontiArc

Both CDs, sequence diagrams and feature diagrams are well known as they are part of the UML [Obj17]. As both models were trained on that data, a better performance can be expected. Therefore, we will attempt to create a model with a lesser known DSL: MontiArc [BKRW17, RRW13] for describing software architectures. They are described as component and connector systems in autonomously acting components that can perform computations. Figure 4.19 shows a simple example: The architecture of a light controller. The model defines how the components are connected to each other and what messages are sent. In this case, a simple controller decides based on a light switch and a door status if it will turn a light on or off. The creation of MontiArc models turned out to be more complex, as the LLMs creates one model for each component (e.g.: one for DoorEval, Arbiter and LighCtrl) and these components have to use consistent ports and connections to each other. Using custom instructions to GPT yielded no results, in contrast to using few-shot learning. With the inclusion of example models within the context, we could reach acceptable success rates with GPT-4 (see Table 4.10), GPT-3.5, however, is not yet able to produce models reliably. Additionally, GPT-4 is able to follow the requirements much closer than GPT-3.5 creating almost all iterations 6 interconnected MontiArc-components within their distinct models, whereas GPT-3.5 amount of components range from 1-7, averaging out at 4,48 and showing significant less consistency.

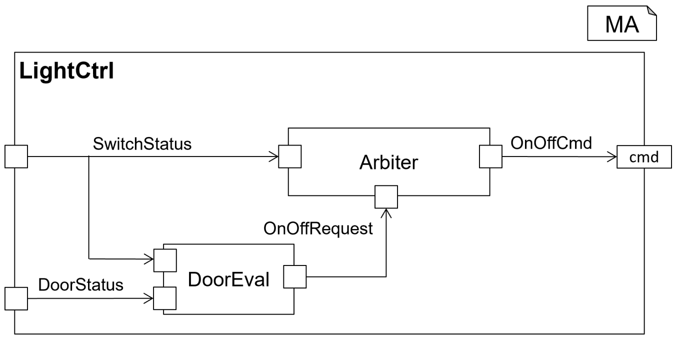


Figure 4.19: Example of a MontiArc (MA) diagram. It defines the architecture for a light controller that reacts on the input of a light switch and whether a door is opened, it returns a corresponding command.

Test Name	Model	Models containing Syntax Errors (N)[1]	Test runs [2]
Hydraulic Press 1-Shot (a)	GPT-4	0 % (0)	22
Hydraulic Press 1-Shot (a)	GPT-3.5	96.94 % (95)	98
Hydraulic Press 1-Shot (b)	GPT-4	100 (30)	30

1: Absolute number on syntactically incorrect models. 2: Absolute number of created models.

Table 4.10: Percentage of invalid models produced with LLMs. GPT-3.5 denotes gpt-3.5-turbo-0613, GPT4 denotes gpt-4-0613.(a) denotes the example artifact "bumperbot" (*cf.* Listing A.8) and (b) the "elevator" example (*cf.* Listing A.7)

The data presented in Table 4.10 for the evaluation of MontiArc artifacts show that the GPT-4 model has a lower error rate than the GPT-3.5 model. Figure 4.20 shows a graphical representation. GPT-3.5 is not able to consistently generate correct MontiArc models, and with a shorter "elevator" (*cf.* Listing A.7) example for few-shot prompting, GPT-4 is not able to generate correct artifacts either. GPT-3.5 seems to be unable to generalize the relatively complex MontiArc artifacts from the chat prompt. Surprisingly, with the only slightly larger example artifact "bumperbot" (*cf.* Listing A.8), GPT-4 is able to generate error-free MontiArc models that are almost completely semantically correct.

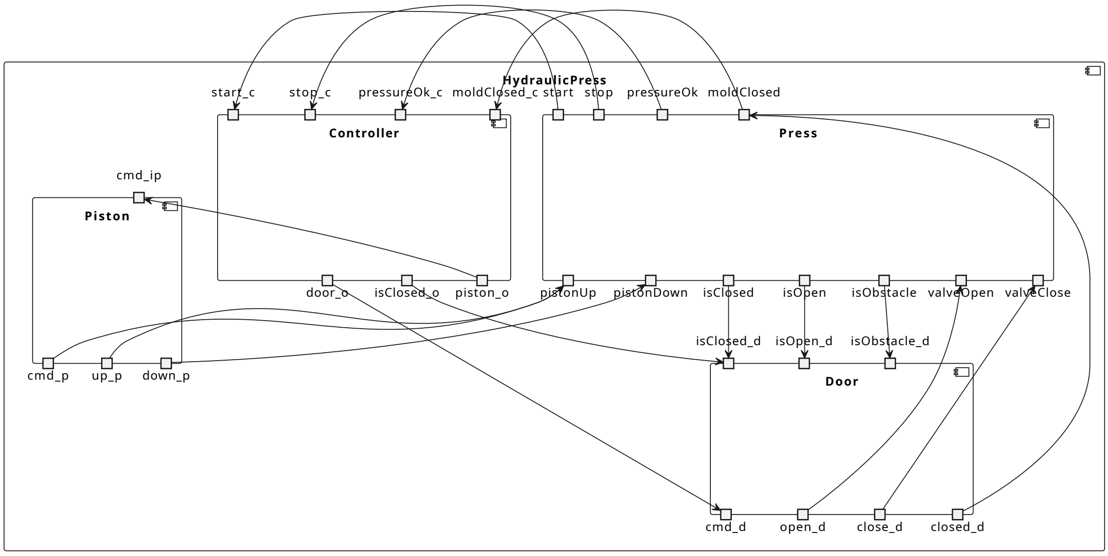


Figure 4.20: Example of a produced MontiArc model for a hydraulic press (*cf.* Table 4.10)

4.7.4 Creating GUIDSL Models

To generate a web application, a user interface description format is necessary. Therefore, we assessed the feasibility of using an LLM to transform into a GUIDSL model. However, up to this point, no valid models could be produced using GPT-3.5 or GPT-4 for any of the user interface descriptions used in this dissertation. Comparable to the difficulties faced in MontiArc, there is limited training data obtainable, and the targeted models have a high level of complexity.
Some success could be achieved by tasking the LLM (GPT-4) with producing very simple user interfaces:

> Task: Create a component with the following specifications: The component has a column with a width of 60%. Within the column, the expression “Welcome” is in the center of a row. Below the row is another column. Within the column there are two new columns that are positioned in the center and have the same horizontal distance to each other. Both columns have a width of 30%. In each of the two columns there is a button in the center of the column. One column has the button ’Start’ and the other column has the button ’Exit’.

Figure 4.21: Prompt defining a very minimal user interface.

All models produced even for this simple UI did deviate from the requirements, and thus did not score the maximum of 17 points. As we need a reliable mechanism to consis-

	Requirements	Maximum score
1	Column 1 (Width: 60%)	2
2	Row: (vAlign und hAlign: centered)	3
3	String „Welcome“	2
4	Column 2 (vAlign: centered, hAlign: even spaced)	3
5	2 Columns within row 2	2
6	Width: 30%	1
7	vAlign und hAlign: centered	2
8	Start-Button	1
9	Exit-Button	1
	$\sum$	17

Table 4.11: Grading schema used for semantic analysis.

Temperature	1	2	3	4	5	6	7	8	9	$\sum$
$t = 0$	2	3	2	0,85	2	1	1,85	1	1	14,7
$t = 0,2$	2	2,97	2	0,43	1,96	1	1,55	0,92	0,92	13,75
$t = 0,4$	1,98	2,63	1,87	0,47	1,71	1	1,42	0,79	0,79	12,66
$t = 0,6$	2	2,13	1,89	0,35	1,82	1	1,43	0,74	0,74	12,1
$t = 0,8$	2	2,02	1,82	0,41	1,87	1	1,54	0,66	0,66	11,98
$t = 1,0$	2	1,97	1,57	0,39	1,69	0,97	1,38	0,63	0,63	11,24

Table 4.12: Average points scored (*cf.* Table 4.11) per requirement for the temperature settings $t = \{0, 0.2, 0.4, 0.6, 0.8, 1\}$, whereby the results are rounded to two decimal places.

tently produce user interfaces, we will proceed with a deterministic, model-driven and transformer-based approach. That reliably derives user interfaces from class diagrams.

Chapter 5

Defining GUI models for Information Systems

To generate an information system we need a modeling language that can define user interfaces. In the following we will take a closer look at two variants of such an DSL: GUIDSL v1 and GUIDSL v2:

Contents

A DSL, or Domain-Specific Language, is a specialized programming language designed for a particular domain or industry [FL10]. It is tailored to address specific problems or tasks within that domain, making it more efficient and easier to use for those purposes compared to general-purpose programming languages. A DSL typically has a limited vocabulary and syntax that directly reflects the concepts and operations of the target domain, enabling users to express their intentions more clearly and concisely [FR07]. Examples of commonly known DSLs include SQL for database management, HTML for web page structure, and CSS for web page styling. Within this work, we focus on a model-driven approach that operates on the basis of specific DSLs, that help developers to define a data-centric web information system. To generate an information system, we need several DSLs that define platform-specific models of the various aspects of such a system. Previous work by Roth [Rot17] and Reiß [Rei16] has already established the DSL for class diagrams CD4A, and for tagging. To create an information system, however, we also need a DSL to define the user interfaces. In the following, we will consider two variants of a user interface description language and take a look at how tagging can be applied to information systems (tagging for CD4A).

- **GUIDSL v1**: The first iteration of a modeling language to define user interfaces of web applications (*cf.* Section 5.1).
- **GUIDSL v2**: The revised version of GUIDSL v1 (*cf.* Section 5.2).
- **Tagging Language for CD4A** A DSL used to add annotations and additional information to another DSL. In this work, we focus on the tagging language as it is used for CD4A (*cf.* Section 5.3).

5.1 Defining Graphical User Interfaces with GUIDSL v1

User interfaces play an essential role in the usability and aesthetics of software applications. To simplify the creation and specification of user interfaces (UIs), developers and designers often turn to DSLs tailored for UI description. The most common DSL used for the description of UI is HTML (Hypertext Markup Language), primarily known as the backbone of web content. HTML combined with CSS (Cascading Style Sheets) can be viewed as DSL to describe the web user interfaces [HM02]. Several DSLs are used to define user interfaces for all kinds of purposes and platforms. A generic common modeling language for user interfaces has not yet been established. Some DSLs that reoccur in the literature are IFML [HBAA18], UsiXML [LMPV96, LVM+04], UMLi [DS02], or XIS-Mobile [RdS14]. A comprehensive evaluation of common UI-modeling languages can be found at [MdS15].
In order to define an information system, we need to define its user interface (*cf.* Figure 2.3). The generators described in Chapter 7 and Chapter 8 use UI-description

languages such as GUIDSL v1 to enable the system modeler to define user interfaces. The presented method was implemented for both variants of GUIDSL. In the following, we will take a closer look at the first variant of GUIDSL.

Please note that some explained elements are not bound to the features of the the language described in this project, but rather comes from the interpretation of models within a generator. The generator will be discussed in Chapter 7.

As the creation of user interfaces is a key element in this work, we will discuss the DSLs used to do so. Within the MontiCore language family, GUIDSL is the language used to define User Interfaces. It was first developed in the context of the MaCoCo-Project (Chapter 8) to accelerate the development of GUIs (*cf.* [**?**]) and was later reused in the following MontiGem projects. The first version of GUIDSL (GUIDSL v1) is based on the web page description language developed by Roth [Rot17] for web information systems. This first version has a strong focus on user interfaces for web applications for financial management due to its origin in MaCoCo. The language supports a wide variety of GUI elements, but is limited to the initial use cases.
Key principles of GUIDSL v1 are:

- One model per page: Each page in the targeted web application is represented as one model

- Predefined behaviors: GUIDSL v1 provides a set of predefined functions such as validation, permission checks, or data format transformations.

- Predefined component set: GUIDSL v1 provides a final component set that can not be extended within the model.

Listing 5.1 depicts a very simple GUI-model, defining a simple web page with only a title card and a simple model. Figure 5.1 shows the resulting userinterface.

```
webpage MyPage {
  card {
    head {
      label "My Page"
    }
    body {
      label "Just a simple Text"
    }
  }
}
```

GUI-DSL v1

Listing 5.1: Simple Example of a GUI-model, defining a simple Web page with a title card and a button. The results are shown in Figure 5.1

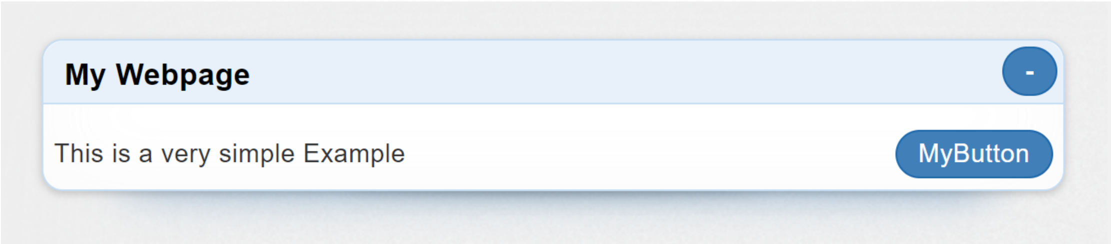


Figure 5.1: Resulting Web page, based on the GUI-model Listing 5.1

The grammar of GUIDSL v1 has to be extended each time new GUI elements are needed. This leads to either GUI elements not being used optimally, as a change requires an adjustment of both the grammar and the generator, or to adjustments being made to the handwritten code. Although GUIDSL v1 is a very capable DSL (it is still in use in the MaCoCo project) a refined version was developed to provide the application developer with a more dynamic DSL that allows a higher degree of customization and extension of GUI elements: GUIDSL v2. In the following, we will take a closer look at both DSLs.
The MontiCore-based language GUIDSL v1 is used to define the user interfaces of a web application. Therefore, it relies mainly on arranging and configuring a predetermined set of GUI elements for each web page of the application. The DSL itself does not define the look and feel of the application. One GUI-model relates to one web page. In the following, we will refer to that web page simply as the 'page'. Figure 5.2 shows the structure of GUIDSL v1. `GUIDSLCore` defines the core structure of a page and provides key interfaces for the elements contained. The page itself and the elements contained can be configured with non-terminals that are defined in `GUIDSLConfiguration`. GUIDSL v1 provides a multitude of GUI-Elements, to define the content of a page. The elements are grouped within seven grammars (charts, tables, input and output, layout, MaCoCo-

specific elements, and navigation) each extending `GUIDSLCore`. Finally, an additional Grammar `GUIDSL` is added to unify the GUIDSL with one grammar.

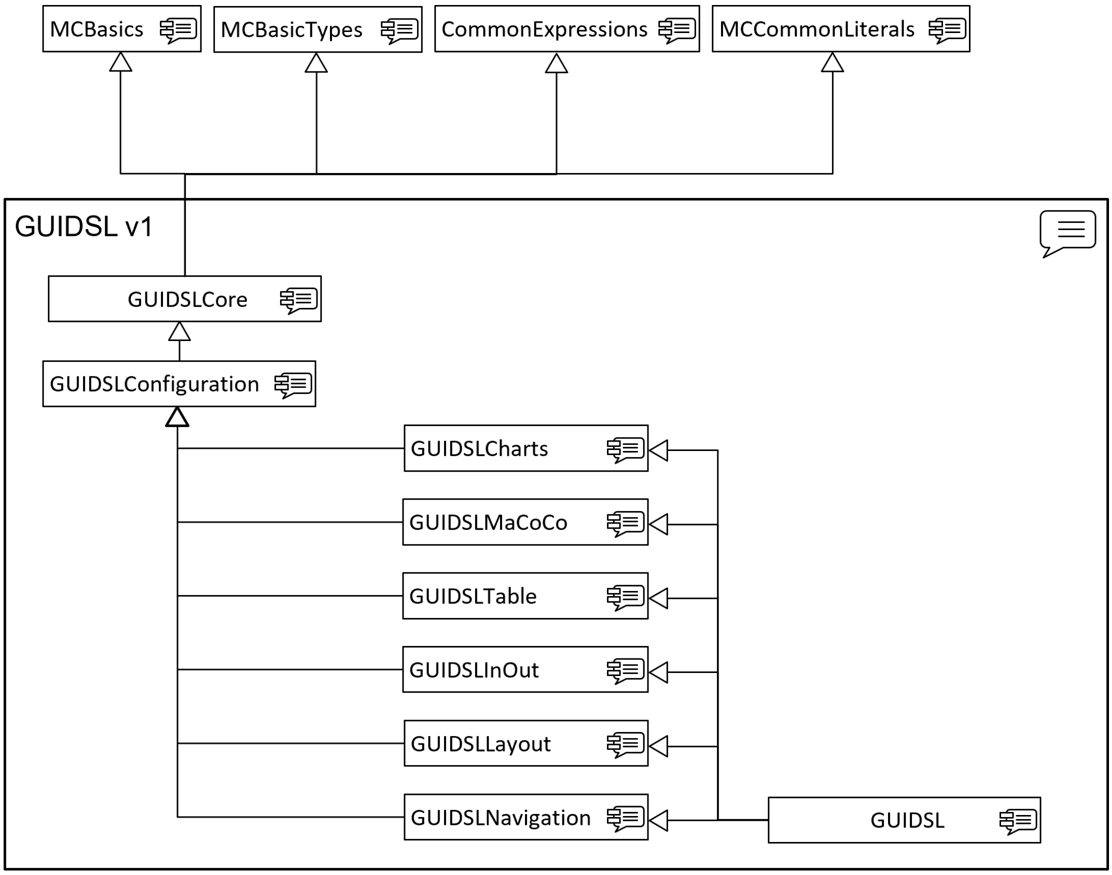


Figure 5.2: Structure of the component grammars of GUIDSL v1. GUI emblems used in GUIDSL v1 are grouped within grammars. Use case-specific extensions can be added by adding another component grammar - see `GUIDSLMaCoCo`.

5.1.1 Core Grammar

GUIDSL v1 is primarily used to define web pages. Within the component grammar `GUIDSLCore`, we define the basic structure of one page. Every GUI-model in GUIDSL v1 consists of the definition for one page (*cf.* Listing 5.2).

```
scope Page implements Root = "webpage" name:Name PageSettings?
  "{" PageElement* "}";

interface PageElement;
```
MCG

Listing 5.2: Production for 'Page' within GUIDSLCore

Each `page` has a set of input parameters with which it is instantiated (*cf.* Listing 5.3). Next to a set of `PageElements`, a page also has a set of `PageSettings`. A `PageSettings` is defined by an input which can be either a `Parameter` or a `FormularInput`.

```
  PageSettings = "(" ( Parameters )+ ")" ;
  Parameters = param:Inputs+;
  Inputs = (Parameter | FormularInputs );
```
MCG

Listing 5.3: Page settings within `GUIDSLCore`

The nonterminal `Parameter` defines one or many values that are loaded via a command once the page is rendered. We distinguish the following :

a) **ById:** Data is retrieved by sending a command to retrieve one object for a specific ID (`byId`)

b) **ByMoreId:** Retrieving one or many objects using multiple IDs (`byMoreIds`)

c) **All:** Sending a command to load `all` objects of one type. In contrast to the ID-specific data retrievals, this mode is also used in case no ID is needed to load data. For example, if there is always only one object of the given type in the database, or the specification of the object is not determined on the client side.

d) **Socket:** Retrieving data as a constant data stream (`socket`)

In addition to parameters, `FormularInputs` can be defined. In contrast to Parameters, they are used to store and send information the user enters in the user interface, and thus operate in both communication directions. In case we define an input field of an editable table, the data would be cached in a `FormularInput` before being sent to the server.

```
symbol Parameter =
  (["all"] | "byId" | byMoreIds:"byMoreIds" | socket:"socket")
  ("(" idName:Name ")")?
  type:MCObjectType
  Name&;

symbol FormularInputs = ("formular")
  formName:MCObjectType "<" genericName:MCObjectType ">" Name&;
```

Listing 5.4: Page input definition within GUIDSLCore

GUI Page Elements

As shown in Figure 5.2, GUIDSL v1 is split up into several component grammars, several of which group the GUI element definitions by category. In the following, we will take a look at a few of the GUI elements that are defined in each component grammar:

5.1.2 Charts

This grammar contains a set of the most common [WBEJ20] chart types: Pie charts, bar charts, and line charts. Page elements defined in this grammar are:

- **`PieChart`** A circular graph that displays data as a proportional part of a whole, with each slice representing a percentage or fraction of the total.
- **`LineChart`** A graph that displays data as a series of points connected by straight lines, used to show trends and patterns over time or other continuous variables.
- **`BarChart`** A graph that uses rectangular bars of varying heights or lengths to represent and compare different categories or discrete data points
- **`HorizontalBarChart`** Same kind of diagram as a bar chart, but with a horizontal orientation.
- **`TimeLineChart`** A visual representation of events or milestones chronologically displayed along a linear scale, often used to illustrate historical or project-related information.

As the page elements above follow similar principles in their definition within the grammar, we will take a look at only two of them: The *pie chart* and the *bar chart*. Within the `GUIDSLCharts` grammar component, a pie chart is defined as follows: `data` defines the list of input values that make up each category within the pie chart. In addition, we have to define which elements of the data list refer to the title of the category and which refer to the actual value.

```
PieChart implements PageElement = "pie" name:String "{"
  "data"  (dataRef:MyClass) "{"
    ("title" ":" title:MyClass)
    ("value" ":" value:MyClass)
  "}"
  PieTotal?
"}";

PieTotal =
  "total" "{"
    "title" ":" title:MyClass
    "value" ":" value:MyClass
  "}"
;
```

MCG

Listing 5.5: Exerpt from GUIDSLCharts definig a PieChart Page element. The pie chart is filled with data from 'MyClass' objects.

Listing 5.6 shows a GUI model using a pie chart (*cf.* Figure 5.3). In order to display values as a pie chart, we need to define a page (Line 1 `MyPage`) and configure the origin of the data we want to use (Line 2 `all MyClass myClass`). In this case, all objects of the type `MyClass` are requested from the server. We define the new pie chart `pie "myPie"` providing it with the entries from the received class. We use the attribute `name` of each entry to define the title of a category in the pie chart and the value attribute to define the corresponding value. We use `euro` to configure the pie chart to display the value as a financial parameter.

```
1  webpage MyPage(
2    all MyClass myClass
3  ) {
4    pie "myPie" {
5      data <myClass.entries {
6        title: <name
7        value: euro(<value)
8      }
9      total {
10       title: <totalName
11       value: <totalValue
12     }
13   }
14 }
```

GUI-DSL v1

Listing 5.6: Example of a GUI model displaying a pie chart

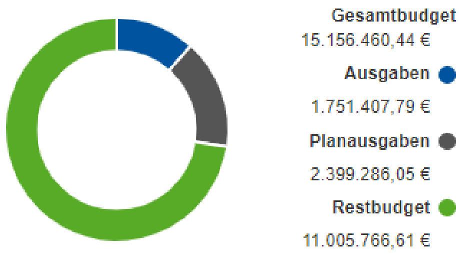
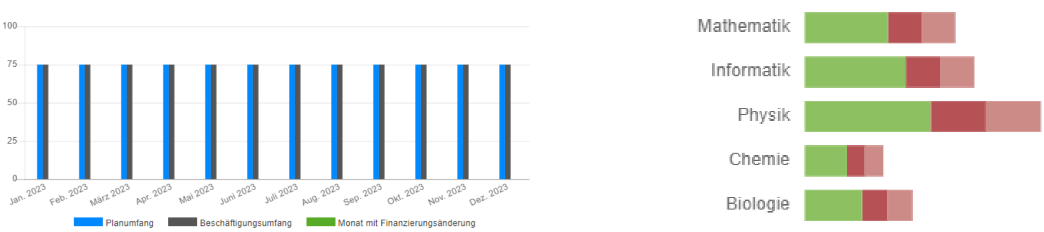


Figure 5.3: An excerpt of a generated user interface showing a pie chart component (Screenshot from the MaCoCo use case). Note that next to the diagram itself, a legend with the raw data is produced as well.

```
BarChart implements PageElement =
  "barchart" "{"
    "data" ":" data:Reference
    "dateRange" ":" dateRange:Reference
  "}" ;
```

Listing 5.7: Exerpt from GUIDSLCharts definig a bar chart page element.



(a) Generated Vertical Bar Chart

(b) Generated Horizontal Bar Chart

Figure 5.4: Examples of bar charts generated with GUIDSL v1

5.1.3 Tables

Tables (*cf.* Figure 5.5) are an important component of any information system. As many use cases of information systems emerge around the interaction with the table component, the list of supported functions has grown over time. Therefore, it is one of the more complex page elements of the GUIDSL grammar. The basic structure is shown

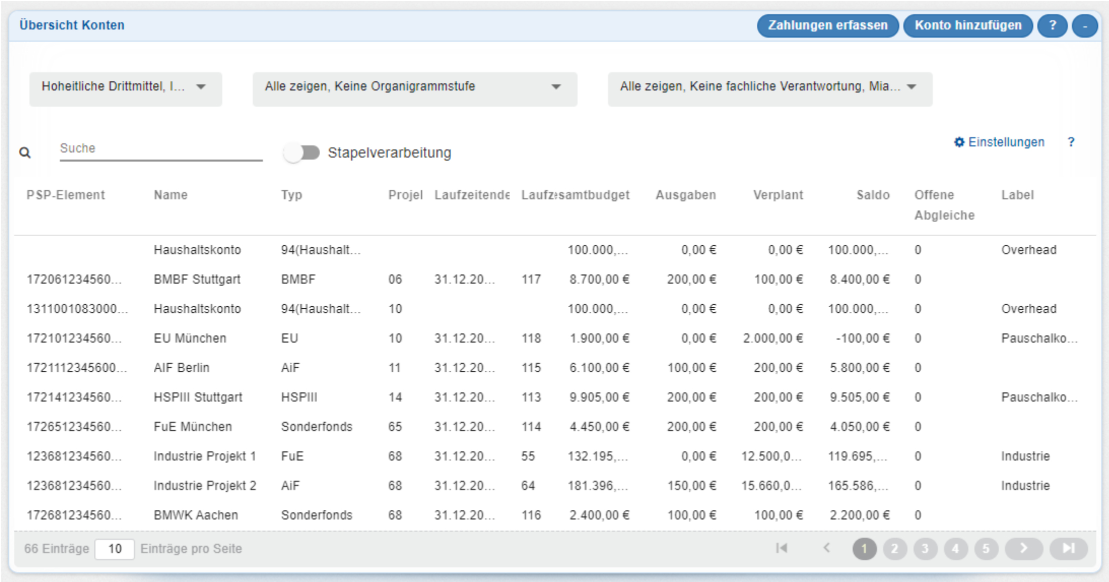


Figure 5.5: A data-table-component as used in MaCoCo. Note that the component does not only provide plain data but also offers a variety of additional functions to search, filter, and process the shown data.

in Listing 5.13

```
MacocoDataTable implements PageElement =
  "datatable" name:String "{"
    DataTableMethods?
    DataTableConditions?
    Content
  "}"
;
```
MCG

Listing 5.8: Excerpt from GUIDSLTable defining a data table page element.

The GUI component of the table (Listing 5.8) is primarily defined by three elements. (1) Functions that are called when interacting with the table can be referenced via `DataTableMethods`. (2) Configurations such as editing capability, a grouping of entries, or export capability can be set through a `DataTableCondition`, and finally, (3) the table content itself is defined through a `Content` production.
The table generated by the GUI generator already supports many interactions. With the `DataTableMethods` production we can define which method (e.g. a method call overwritten by hand) should be called if one function of the table is used. Listing 5.9 displays some of the implemented functions of a table. `ViewMethod` (Line 4, Line 13) is called

if a user clicks on a table row to inspect it. Similarly `EditMethod` is configured if the user chooses to start editing a row. `SaveMethod` and `DeleteMethod` are typically used to define a method that transmits the changed or removed row to the server and stores the updated data.

MCG

```
DataTableMethods =
  "methods" "{"
    (
        ViewMethod
      | EditMethod
      | DeleteMethod
      | SaveMethod
      // [...]
    )*
  "}"
;

ViewMethod implements DataTableMethod = "view" "->" Expression;
EditMethod implements DataTableMethod = "edit" "->" Expression;
DeleteMethod implements DataTableMethod = "delete" "->" Expression;
SaveMethod implements DataTableMethod = "save" "->" Expression;
// [...]
```

Listing 5.9: Excerpt from GUIDSLTable defining method productions of a data table page element.

The functionality and appearance of the table can be configured with `DataTableConditions`. A condition can be, for example, `NoFooterCondition`(Lines 3, 12), which disables the footer of the table containing a paging mechanism and a current row count, comparable to `NoHeaderCondition`(Lines 5, 16), which disables the header of the table. The condition `GroupableCondition`(Lines 4, 14), allows the user to group elements in the table. The developer can specify by what elements the table is grouped, by default setting `groupableRef`. We can configure the table to allow the user to add new entries by setting the `HasCreateNewCondition`. In total, there are 21 conditions that can be set in order to configure the behavior and appearance of the table components. As the table is a key component of many projects, further configurability was added in order to ensure that the table meets all requirements of multiple use cases. Conditions can be constrained by `PageElementConditions`, which in turn can be defined by Boolean parameters or methods, or permission settings.

```
DataTableConditions =  "conditions" "{"                          MCG
  (
    NoFooterCondition
    | GroupableCondition
    | NoHeaderCondition
    | HasCreateNewCondition
    // [...]
  )*
  "}"
;

NoFooterCondition implements DataTableCondition =
    "noFooter" PageElementCondition?;
GroupableCondition implements DataTableCondition =
    "groupable" (":" groupableRef:Reference)? PageElementCondition?;
NoHeaderCondition implements DataTableCondition =
    "noHeader" PageElementCondition?;
HasCreateNewCondition implements DataTableCondition =
    "hasCreateNew" PageElementCondition?;
// [...]
```

Listing 5.10: Excerpt from GUIDSLTable definig a condition productions of a data table page element.

Finally, after defining what features should be enabled and setting the methods behind the functionality, we can define `Content` (Listing 5.11) of the table. A table consists of `rows`. A list object is referenced containing the entries of the table. The columns (`Col*`) define what attributes of each entry in the list object should be displayed in which manner (*cf.* Listing 5.13). Each `column` defines how an attribute is shown in the table, defining elements such as the width or column title. Listing 5.12 shows the definition of a table column. Similarly to the conditions of the entire table, columns can also be constrained by a `PageElementCondition` (Line 2) each column has a name and refers to one attribute (`TableCell`). Additionally, a column itself can have some conditions. Among other options it can be made `mandatory`, preventing this field from being left blank upon saving. Other options can be `hidden`, thus not displaying an attribute, but leaving an option to the user to display it again. Or `disabled`, showing this attribute, but preventing the user from editing it.

MCG

```
Content =
  "rows" rowName:Reference "{"
    Col*
  "}"
;
```

Listing 5.11: Excerpt from GUIDSLTable defining the content productions of a data table page element.

MCG

```
Col =
  "column" PageElementCondition? (columnName:String) ","
      tableCells:TableCell ("," width:NatLiteral)
  (
    ("," ["mandatory"])
    | ("," ["hidden"])
    | ("," ["disabled"])
  // [...]
  )* ";"
;
```

Listing 5.12: Excerpt from GUIDSLTable defining the column productions of a data table page element.

In the following, we will take a look at an example of a GUI model that defines a table component. Listing 5.13 displays a web page `TableExample`. Within the page the page element `myExampleTable` is defined. For this table, one method is defined (Line 4). If the user clicks on a row, method `navigateToDetails()` is called. Additionally, two conditions are set: `deletable` and `viewable`. Enabling the user to click on rows and to delete entries. The table has three columns: `Name`, `Adress`, and `Date of birth`. All of these are attributes of the `tableEntry` object that is used as data input for the table. Similarly to Listing 5.6 we can use `date` to display the corresponding attribute as a time parameter (e.g. giving it a date picker upon editing).

```
webpage TableExample(all TableObject to) {                    GUI-DSL v1
  datatable "myExampleTable" {
    methods {
      view -> navigateToDetails($event)
    }
    conditions {
      deletable
      viewable
    }

    rows <to.tableEntry {
      column "Name"   ,           < name                      ,  80;
      column "Address"        ,  < address                   , 120;
      column "Date Of Birth" ,  date(< dateOfBirth)         , 100;
    }
  }
}
```

Listing 5.13: GUI model defining a data table. The table starts with the keyword `datatable`, has one `method`, defining the method that is called upon a click on a row. Next, `conditions` of the table are defined, making the entries clickable (`viewable`) and `deletable` finally, the `rows` are defined by configuring three `columns`.

5.1.4 Input & Output Elements

The component grammar GUIDSLInOut defines key interactive elements in the user interface. The page elements defined in this grammar are:

- **`Image`** A simple page element that embeds an image into the page.
- **`Label`** A simple text.
- **`Infosign`** A colored box with an icon, a title, and a text is used to provide the user with critical or useful information.
- **`Button`** A simple button with different styles.
- **`HelpButton`** A customized version of `Button` but only with a question mark and only with links to the handbook pages.
- **`SelectButton`** A toggle button.
- **`TextInput`** An element that allows users to input and edit multiple lines of text, typically used in web forms or for user-generated content.

- **`CheckBox`** An element that allows users to select or deselect an option, typically used for multiple choice questions or to indicate agreement to terms and conditions.

- **`RadioButton`** An element that allows users to select one option from a group of mutually exclusive options, typically used for single-choice questions or selecting preferences.

- **`TextArea`** An element that can display multiple lines of text.

- **`DropDown`** A graphical user interface control that allows users to choose from a list of predefined options, typically displayed as a list that expands when clicked or hovered over.

- **`AutoComplete`** Similar to `DropDown`, but with the added functionality of autocomplete which displays suggestions as the user types, helping to speed up the selection process.

In the following two of these components will be presented: *Infosign* and *Button*.

Infosign

The info sign shown in Figure 5.6 is used to provide guidance, instructions, or information to users on the website. It is used to communicate important messages quickly and effectively, improving safety, efficiency, and usability. Examples of info signs include directional signs, warning signs, regulatory signs, and informational signs. In GUIDSL v1 (Listing 5.14) `Infosign` is a page element that, similar to most page elements, can be shown or hidden based on an optional `PageElementCondition` (Line 2). The element is defined by an icon, either in `info`, a spinning `load` icon, or a custom-defined one. In addition, a title (`header`) and a `message` have to be provided.

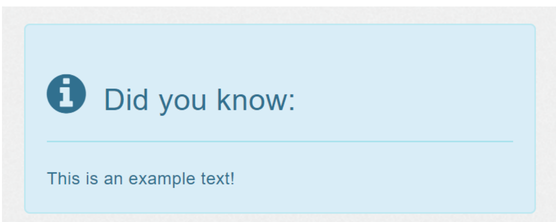


Figure 5.6: Example of an info sign as used in MaCoCo

MCG

```
InfoSign implements PageElement =
  "infosign" PageElementCondition? "{"
    "icon" ":" (["info"] | ["load"] | icon:String)
    "header" ":" (header:String)
    "message" ":" (message:String)
  "}"
;
```

Listing 5.14: Excerpt of the grammar GUIDSLInOut defining an info sign page element

GUI-DSL v1

```
infosign{
  icon    :  info
  header  : "Did you know"
  message : "This is an example text!"
}
```

Listing 5.15: Example of a GUIDSL v1 model defining the info sign page element shown in Figure 5.6

Button

The generic button is a key component in every application. Therefore, GUIDSL provides a button that can be customized in multiple ways. Listing 5.16 shows the corresponding excerpt from the GUIDSLInOut grammar. `Button` is a `PageElement` that can be hidden by a `PageElementCondition`. The button itself has a `label`, which defines the displayed text. A button can have many configurations, some of which are listed in Listing 5.16 line 4..8. Specific style classes can be set at `ConfigurationStyleclass`, the button can be disabled via `ConfigurationDisabled` height and width can be set via `ConfigurationHeight` and `ConfigurationWidth`. Finally, the method that should be called upon clicking the button can be set to `ButtonClick`. Note that instead of a method call, the method body can also be provided (line 14). An example model is shown in Listing 5.17. In Line 1 the button is defined to display the text `"Button A"` and to only be visible if the method `showButton()` returns `true`. Further, we define the method that is called upon clicking the button: `navigateToOverview()` (line 2) and define the dimensions of the button in Lines 3..4: `height: 30` and `width: 300`.

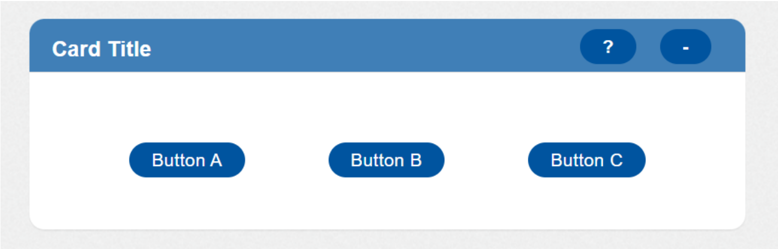


Figure 5.7: Example of three buttons as used in MaCoCo

MCG

```
Button implements PageElement =
  "button" PageElementCondition? (label:String) "{"
    (
        ConfigurationStyleclass
      | ConfigurationDisabled
      | ConfigurationHeight
      | ConfigurationWidth
      | ButtonClick
      //[...]
    )*
  "}"
;

ButtonClick = "click" "->" click:Expression ("{" methodBody:String "}")
    ?;
```

Listing 5.16: Excerpt of the grammar GUIDSLInOut defining an button page element

GUI-DSL v1

```
button if(showButton()) "Button A" {
  click -> navigateToOverview()
  height : 30
  width : 300
}
```

Listing 5.17: Excerpt of an GUIDSL v1 model defining an button shown in Figure 5.7

5.1.5 Layouting

The component grammar `GUIDSLLayout` defines page elements that handle the arrangement of other page elements. This grammar contains:

- **`Container`**: Simple layout element that can be used to box in page elements

- **`Column`**: Similar to a container, but arranges contained elements in a column.
- **`Row`**: Unlike a container, it arranges the contained elements in a row.
- **`Card`**: A page element that displays a rounded box with a header. Elements in both the header and the body are arranged in a row.

Card

Let us take a look at the `card` page element shown in Listing 5.18. A card consists of two subcomponents (Line 5,6), an optional `CardHead`(Line 10), and a `CardBody` (Line 12). The `head` is used to display a title. It can be used to add buttons that refer to the content of the body of the card. The cards are collapsible (using the "-" button shown in Figure 5.8 and Figure 5.7. Cards are primarily used to encapsulate and organize content within the generated web pages.

```
MCG
1  Card implements PageElement =
2    "card" name:String? PageElementCondition?
3    ("collapsedIf" "(" collapsedIf:Expression ")")?
4    "{"
5      head:CardHead?
6      body:CardBody
7    "}"
8  ;

10 CardHead = "head" ("color" ":" color:String)? "{" PageElement* "}";

12 CardBody = "body" PageElementCondition? ("color" ":" color:String)? "{"
       PageElement* "}";
```

Listing 5.18: Excerpt of the grammar GUIDSLLayout defining an card page element

```
GUI-DSL v1
card {
  head {
    // page elements for the title bar of the card
    textoutput {"Exampletext"}
  }
  body {
    // page elements for the title bar of the card
    button "A Button" {  click -> doSomething() }
  }
}
```

Listing 5.19: Excerpt of a model defining an card page element

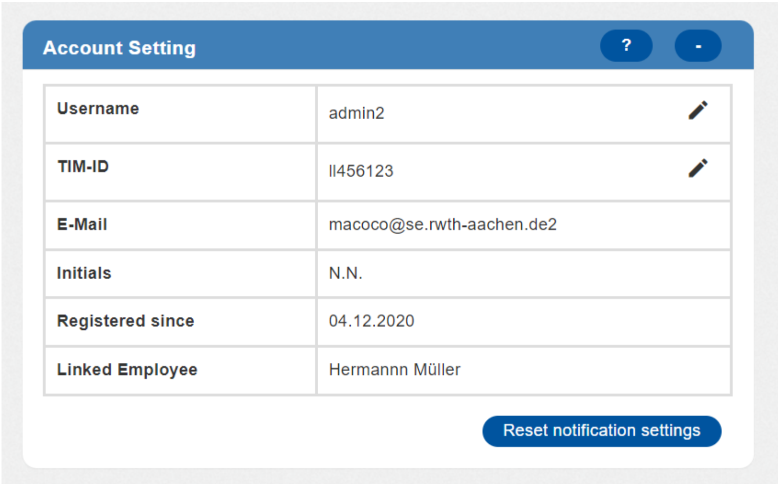


Figure 5.8: Example of the page element card "Account Settings" (containing a simple table) as used in MaCoCo

5.1.6 Navigation

Next to the definition of web pages, GUIDSL v1 can be used to define a navigation tree for the entire website. As this tree of page references can be used to generate menus and sidebars for page navigation. The grammar defines the following elements:

- **`NavigationItemArray`**: A nested list of `NavigationItems` that can be used to define a navigation menu.
- **`NavigationBar`**: Page Element that can be used to display navigation elements within a page.

NavigationItemArray

Within the GUIDSL v1 we can use `NavigationItemArray` as shown in Listing 5.20. In order to define a navigation menu within our application. A navigation menu is a user interface element that displays a list of links or options that allow users to navigate to different sections or pages within a website or application, as shown in Figure 5.10. It typically appears on the top or side of a web page and may be organized hierarchically to help users find the information they need more easily. Navigation menus can include links to the homepage, contact page, about page, product or service pages, and any other important pages or sections of the website. A `NavigationItem` is defined by the Link (`NavigationItemLink`) it navigates to, the text (`NavigationItemLabel`) and icon (`NavigationItemIcon`) it displays, as well as further nested items (`NavigationItemChildren`) and conditions (`NavigationItemCondition`) similar to `PageElementConditions` to define when an element should be visible.

MCG

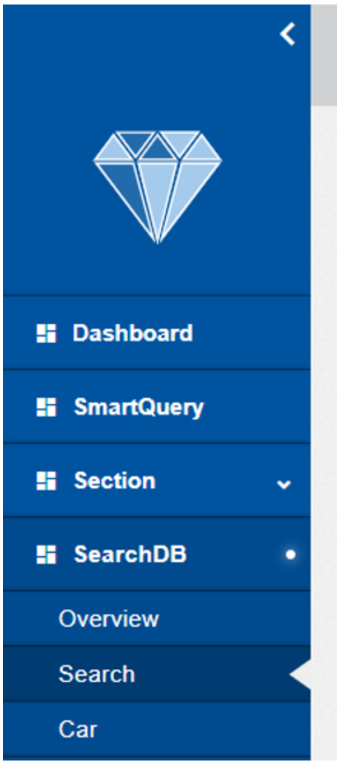
```
Navigation implements Root = "sidenavigation" ":" NavigationItemArray;

NavigationItemArray = "[" (NavigationItem || ",")* "]";

NavigationItemLinkArray = "[" (String || ",")+ "]";

NavigationItem =
  "{"
    (
        NavigationItemLink
      | NavigationItemLabel
      | NavigationItemIcon
      | NavigationItemChildren
      | NavigationItemCondition
    )*
  "}"
;
```

Listing 5.20: Excerpt of the grammar GUIDSLNavigation defining a `Navigation`. A Navigation consists of a list of Navigation Items (`NavigationItemArray`), each `NavigationItem` can contain a link (`NavigationItemLink`) a text to display (`NavigationItemLabel`), an Icon (`NavigationItemIcon`), further Navigation Items (`NavigationItemChildren`), and my own conditional (`NavigationItemCondition`)

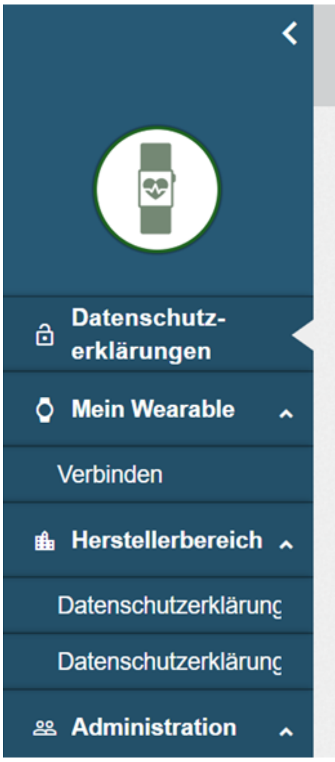
GUI-DSL v1

```
sidenavigation : [
  {
    label: "Dashboard" link: ["/", "dashboard"] icon: "dashboard"
  },
  {
    label: "Finance" link: ["/", "finance", "dashboard"] icon: "euro_symbol"
    children: [
      { label: "Accounts" link: ["/", "finance", "accounts", "overview"] },
      { label: "Billing" link: ["/", "finance", "billing"] },
      { label: "Deadlines" link: ["/", "finance", "deadlines"] }
    ]
  }
]
```

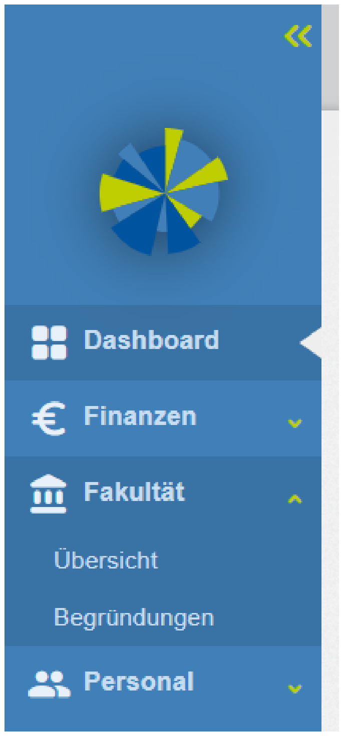
Listing 5.21: Excerpt of the GUI model that defines Navigation shown in Figure 5.10



(a) Default MontiGem



(b) Invidas use case



(c) MaCoCo use case

Figure 5.9: Examples of side bar navigation generated with GUIDSL v1

5.1.7 Context Conditions

GUIDSL v1 does implement very few context conditions in Java but rather relies on `astrules` [HKR21] to ensure valid models. These constraints primarily ensure the uniqueness of configurations within components, e.g. ensuring that there can only be one `ConfigurationWidth` defined for a button (*cf.* Listing 5.16 or that there is not more than one `ViewMethod` defined as a `DataTableMethod` (*cf.* Listing 5.9). The definitions of the page elements make up the bulk of the grammar. These symbols work mostly in isolation and independently of each other. For example, a text field can be inserted anywhere in the model without affecting other symbols. There is little need for context conditions in GUIDSL v1, as they have little context that must be constrained.

5.2 Defining Graphical User Interfaces with GUIDSL v2

The second user interface modeling language for which the presented method was realized for is GUIDSL v2. In the following, we will take a closer look at this DSL.

Please note that some explained elements are not bound to the features of the the language described in this project, but rather comes from the interpretation of models within a generator.

5.2.1 Distinction with GUIDSL v1

In this thesis, we use two distinct DSLs to define graphical user interfaces. The initial version of GUIDSL (*cf.* [GHK+20]), which is rooted in [Rot17], has undergone enhancements and modifications to meet the demands of data-driven web information systems [GMN+20]. Despite its expansive set of component grammars, it remains likely that new components must be incorporated via grammar extensions to address the evolving use-case demands. A notable limitation of GUIDSL v1 is its inability to refer to external models, making the integration of model libraries unfeasible. Given the systematic nature of the user interfaces of web information systems, the deployment of GUIDSL v1 often culminates in repetitive code in multiple models. Subsequently, this redundancy amplifies the maintenance workload to maintain application-wide consistency. GUIDSL v2 enables the system modeler to encapsulate reusable components in separate models that can be referenced. Another shortcoming of GUIDSL v1 is its omission of references to CD4A.

While GUIDSL v1 delineates objects for each element of the page, it fails to perform type checks on associated classes, leaving the validation task to developers and risking potential runtime failures. Despite these limitations, DSL remains adept at articulating user interfaces for comprehensive real-world platforms, as shown by [ANV+18]. The system presented in Chapter 8 is built upon GUIDSL v1.

Having learned from development with GUIDSL v1 we developed GUIDSL v2. Although the name suggests an evolution from one DSL to the other, GUIDSL v2 was a new development and its grammar has little in common with the previous version of DSL. Although both languages are designed to define web pages of enterprise information systems, GUIDSL v2 provides several advantages over GUIDSL v1:

- **Referencability of GUI components:** Components can be defined in one model and be used in another. This allows for the definition of predefined complex components and model libraries, and enables mechanisms to adapt and extend models that are produced by a generator.
- **Referencability of class diagrams:** Used types are defined and checked in a referenced class diagram.

- **Loops:** Components can be rendered based on input data. This enables the definition of dynamic content such as listings of components based on available data (e.g. feeds as known from Facebook or Twitter)

- **Introduction of new components through arbitrary import:** New components can be added by extension of the RTE and by definition of a new GUI component. Components can be imported into other components. There is no need to extend the grammar of the language.

5.2.2 Basic Structure

GUIDSL is a modeling language, developed primarily to describe graphical user interfaces of web-based information systems. It is used primarily in the context of data-centric information systems, as is the case in this thesis; however, GUIDSL is not limited to that domain. Successful experiments in other domains and target frameworks (Mobile Platforms, Flutter[1], React[2]) were performed for both GUIDSL v1 and GUIDSL v2.

```
package example.simple; // package declaration                 GUI-DSL v2

import example.MyComponent; // import statements

page MyPage(String username ) { // GUIComponent declaration

  // ... components, expressions, guards ...

}
```

Listing 5.22: Basic GUI-model structure. Each model has a `package`, can define imports and defines a `page` or a `component`.

A GUI-Model defined with GUIDSL v2 (*cf.* Listing 5.22) consists of a package declaration, import statements, and a `GUIComponentDeclaration`. This component declaration defines a GUI component that can either be a page, used on its own to represent a web page in the final product, or a component used as parts of other components or pages in order to be able to reuse parts of models. The grammar of this looks as follows:

[1](Only GUIDSL v2) `flutter.dev`
[2](Only GUIDSL v2) `react.dev`

```
MCPackageDeclaration

MCImportStatement*

//GUIComponentDeclaration starts here
"page" | "component" Name "("
    GUIParams?
")"

GUIBlock?
```

Listing 5.23: Simplified GUIDSL grammar

The package is used to import the component or page into other models in order to create composite pages or to define a newly composed component. Similarly to Java, the package name reflects the folder structure. The component declaration has a name and `GUIParams`, these functions are similar to function parameters in Java and are used for component instantiation in another model or for an automatic instantiation if the component is declared as a page. Finally, the `GUIBlock` is where the appearance and function of the page or component are specified.
An example of a page can be found in listing 5.22. A component (`GUIBlock`) can be defined by the following symbols:

- `Expression`
- `GUIComponent`
- `GUIGuardExpression`
- `GUIIterateExpression`

Expressions are based on MontiCore-CommonExpressions, e.g. simple mathematical, literal, boolean expressions, etc. An expression is typically used as an argument for the instantiation of components or represents a variable. In the following, we will take a closer look at the remaining productions.

5.2.3 GUIComponent

The GUI component allows the placement of other components on the page. We distinguish four types of GUI components.

- **Generic Component**: A software component with a graphical representation in a web page that optionally includes a data state and a connection to a server back end for receiving and updating the data. A GUI component allows visual interaction and/or provides a representation of information in a graphical (diagrammatic or textual) form.

- **Library Component**: A predefined GUI component, such as a button or a text field. These components serve as a foundation for assembling first models or more complex components.

- **Composed Component**: A component is a composition of other GUI components to form a new, more complex component.

- **Container Component**: A GUI component whose primary purpose is the allocation of space, grouping, and arrangement of GUI components within a container. A container is not necessarily directly visible. For example, when GUI components are arranged in a grid pattern, one may consider the grid arrangement a property of a container.

Note that a GUI component can be generic, library, and composed at the same time, since predefined components themselves can be composed already (e.g., GemCard). The grammar rule for the usage of a GUI component usage in GUIDSL v2 is defined as follows:

```
GUIComponent =
  id:GUIVariableDeclaration? "@"
  Name:GUIFunction GUINamedArgs
;
```
MCG

Listing 5.24: GUIComponent definition within GUIDSL v2 grammar

The definition of a GUI component starts with an id:`id` is an optional identifier that can be used to reference the specific instance of the GUI component. Next, the name of the GUI Component is declared after the "@" notation. Finally, a collection of attributes that are passed to the component in the form of assignments (`GUINamedArgs`) are declared. Note that the GUI Component symbol has both a `name` and an `id`, this is necessary as a GUI Component with the same name can be instantiated in one page (or component definition) multiple times (e.g. displaying the same button multiple times in one user interface). The introduction of the ID enables the developer to refer to specific instances of the button. A component can also use other components as arguments to create composite components (e.g. card, grid alignments of components). An example is shown in Listing 5.25.

```
text1@Text(
  value="This is a Text"
)
```
GUI-DSL v2

Listing 5.25: Usage of a GUI component in GUIDSL v2

5.2.4 GUIGuard

The `GUIGuardExpression` has a similar functionality as the if-statement. It has a condition that is evaluated to a Boolean. If the defined condition holds, the `thenStatement` is processed; otherwise the optional `elseStatement` is used.

```
GUIGuard =                                                    MCG
  id:GUIVariableDeclaration? "@guard" "(" condition:Expression ")"
    thenStatement:GUIBlock
  ("else"
    elseStatement:GUIBlock
  )?
;
```

Listing 5.26: Definition of a GUI guard expression in GUIDSL v2 grammar

Similarly to the GUI Component, GUIGuardExpression has an optional `id` that allows giving the resulting statement a variable for later reference. In contrast to the if-statement as defined in Java, the grammar does not specify when and how often the condition is evaluated. The web applications we generate evaluate regularly the condition, leading to pages that react dynamically to the provided data set. An example of the usage of the GUIGuard is shown in Listing 5.27.

```
//...                                                         GUI-DSL v2
component RoomDashboardGuard(Room rooms) { // Component definition
  @guard(!room.occupied) {  // Check if a room is free
    // Show a RoomInfo Component:
    @RoomInfo(
      roomName = room.name,
      occupied = room.occupied
    );
  }
}
```

Listing 5.27: Usage of a GUI guard expression in GUIDSL v2

The Web page shows a component for a room only if the room is not occupied. The component appears as soon as the Boolean 'occupied' of the room switches to `false`.

5.2.5 GUIIterate

The `GUIIterateExpression` allows the creation of components that have an unknown number of sub-components at generation (e.g., a news feed). An example is shown in 5.29. The construct works similarly to the for loop or while loop. Similarly to the `GUIGuard`, it is intended to be evaluated regularly at the runtime of a generated application, and thus allows for dynamically changing numbers of elements. Similarly to the GUI component the `GUIIterate` production, can define an id, to be referenced later. It includes `IterateControl` which defines an iterable and a variable representing an iteration item. For each iteration, the GUI components defined in the `GUIBlock` are instantiated. When the iterable list object changes, the GUI is automatically updated.

```
GUIIterate =                                                   MCG
  id:GUIVariableDeclaration? "@iterate"
  "(" IterateControl ")" GUIBlock
;
```

Listing 5.28: Definition of a GUI iterate expression in GUIDSL v2 grammar

```
page RoomDashboard(List<Room> rooms) {                  GUI-DSL v2
  @iterate (Room r : rooms) {
    @Text(value = r.name);
  }
}
```

Listing 5.29: Example of the `iterate`, iteratively instantiating a text component in a GUI, and thus creating a list of rooms.

Similarly to GUIGuard, GUIDSL evaluates GUIIterate regularly updating it any time the data changes.

5.2.6 Context Conditions

The following context conditions are defined for GUIDSL v2.

- **`GUIParamDefaultFitsType`:** Checks if the default type matches the declared type.
- **`GUIRangeIterateControlExpressionsAreInt`:** Checks if the range of the `iterate` component is declared as `Integer`.
- **`GuardConditionIsBoolean`:** Checks if the condition of a `guard` component is evaluated to a `boolean`.

5.2.7 Library Components

Library components are the most basic building blocks of the GUIDSL v2. The language is provided with a collection of these predefined components, enabling the developer to compose user interfaces. In the following, we will take a look at some of those predefined components. Note that this provided component library can be extended by custom components for any given use case.

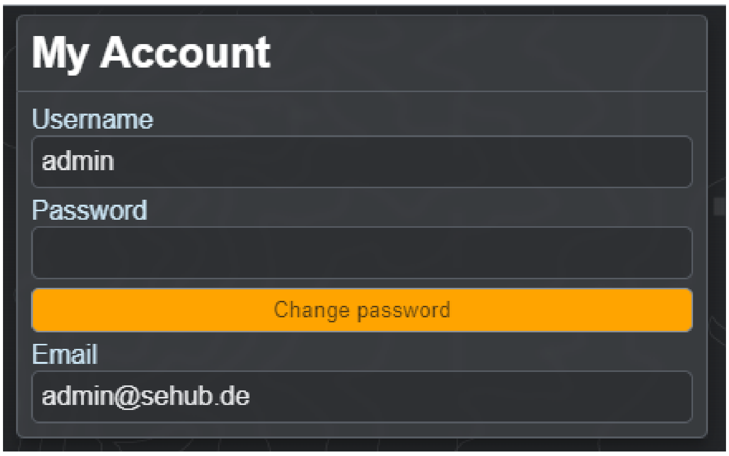


Figure 5.10: Screenshot of a UI (SEHub) defined by Library components (`Button`, `TextInput`, `Column`)

Button

The button is one of the most important interactive GUI elements. The provided library GUI component (*cf.* Listing 5.30) is defined by a String `text` and an `Event`, `leftClick`. The `Event` type represents an event in a system, typically triggered by a user, such as a left click or a hover. In the button component, `Event` is provided when launching the GUI component and is executed when the button is clicked. The type parameter provides (`Void` in the button component) a hint on the type of object that will be passed when an event is triggered, allowing an application developer to define a specific behavior.

```
package example.atomic;                                    GUI-DSL v2

component Button(
  String text = "",
  Event<Void> leftClick
)
```

Listing 5.30: Button GUI component definition

```
@Button(text = "MyButton", leftClick = doSomething)        GUI-DSL v2
```

Listing 5.31: Usage of the Button component in a GUI model

TextInput

The `TextInput` component produces a simple text field. The component can be configured with an optional `String value`, a `label`, and a `placeholder`. A great benefit of the GUIDSL v2 language compared to GUIDSL v1 is its ability to refer to other models. In the following example, the `TextInput` component uses the `username` attribute, to store any input.

```
package example.atomic;                                    GUI-DSL v2

component TextInput(
  ?String value,
  String label = "",
  String placeholder = ""
)
```

Listing 5.32: TextInput GUI component definition

```
@TextInput(                                                GUI-DSL v2
  value = username,
  label=  "MyInput",
  placeholder=  "write here"
)
```

Listing 5.33: Usage of the TextInput component in a GUI model

Layout

There are a few library GUI components that are used to arrange and group other components. Within GUIDSL v2 there are the following predefined container components: `Column`, `Row`, `Grid`, and `Card`. We will take a look at the `Column` component Listing 5.34 Similar to the previously shown components the `Row` component is defined via a set of parameters. This component can be configured via `hight`, `width`, `hAlign`, `vAlign`, and a style class `style`. In addition to these parameters, the component holds a list of GUI Components (`List<GUIViewElement> components`). These elements will be arranged in a column within the generated user interface. Listing 5.35 displays the usage of the column component.

GUI-DSL v2

```
component Column (
  String height,
  String width,
  String hAlign,
  String vAlign,
  String style,
  List<GUIViewElement> components,
)
```

Listing 5.34: Column GUI Component Definition

GUI-DSL v2

```
@Column(height = "500px", hAlign = "center", components = [
// ... components ...
]);
```

Listing 5.35: Usage of the Column component in a GUI model

5.2.8 Defining a Model in GUIDSL v2

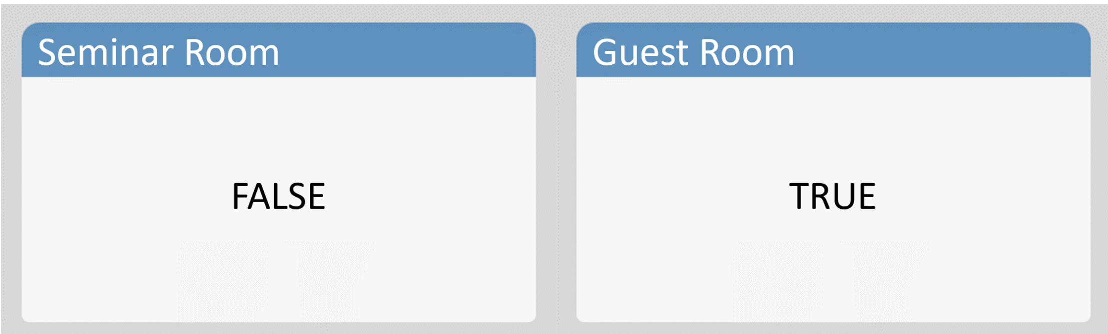


Figure 5.11: Dashboard showing a card each for the Rooms `Seminar Room` and `Guest Room`. The card content indicates if the room is occupied or not, as defined by Listing 5.36

In the following, we will define a simple web page with a set of GUI models as shown in Figure 5.11. The page is defined in Listing 5.36, by configuring a `page` (Line 7). The page represents a simple dashboard that indicates whether two rooms are occupied or free. The page is provided with two `Room` objects: `room1` and `room2`. These objects are used in the two `RoomInfo` components `r1` (Line 10) and `r2` (Line 16). These components are provided with the name and the occupied status, as well as an action that is called upon a click on the component.

GUI-DSL v2

```
1  package pages.room.simple;

3  import example.Domain.Room;
4  import example.simple.RoomInfo;

6  page RoomDashboard(
7    Room room1,
8    Room room2
9  ) {
10   r1@RoomInfo(
11     roomName = room1.name,
12     occupied = room1.occupied,
13     clickRoomInfo = navigateRoomDetails1
14   );

16   r2@RoomInfo(
17     roomName = room2.name,
18     occupied = room2.occupied,
19     clickRoomInfo = navigateRoomDetails2
20   );
21 }
```

Listing 5.36: GUIDSL v2 model defining a simple dashboard showing cards that indicate if a room is occupied or not.

In order for this model to function the `RoomInfo` component has to be imported (Line 4) and defined. Next, we take a closer look at the `RoomInfo` component definition in Listing 5.37. The component `RoomInfo` is defined in the package `example.simple` and imports two additional components: The composed component `Card` and the library `Text` component. The `RoomInfo` component uses as input the name of the room (`String roomName`) its current status (`boolean occupied`) which is default `false` and an event, that can be executed upon clicking the component (`Event<Void> clickRoomInfo`). The component definition contains a `@Card` component that both `roomName` and `occupied` are passed on. Finally, we require a definition of the `Card` component, as shown in Listing 5.38.

GUI-DSL v2

```
package example.simple;

import example.simple.Card;
import example.atomic.Text;

component RoomInfo(
  String roomName,
  boolean occupied = false,
  Event<Void> clickRoomInfo
) {
  @Card(
    header = roomName,
    body = @Text(value = occupied)
  );
}
```

Listing 5.37: Definition of the `RoomInfo` component

GUI-DSL v2

```
1  package example.simple;

3  import example.atomic.Text;
4  import example.atomic.Button;
5  import example.atomic.Column;
6  import example.atomic.Row;

8  component Card(
9    String header = "",
10   GUIViewElement body
11 ) {
12   @Column(width = "30%", components = [
13     @Row(vAlign = "center", hAlign = "spaceBetween", components = [
14       @Text(value = header, color = "white")
15     ]),
16     @Column(vAlign = "center", hAlign = "center", component = body)
17   ]);
18 }
```

Listing 5.38: Definition of the `Card` component

The `Card` component is composed of several layout elements that assemble a box with a header that can contain a label. The component uses as an input a `String` for a `header` (Lines 9 and 14) and a `GUIViewElement body` (Lines 10 and 16) containing the GUI component that is shown within the box. A card is layouted as a `Column` with two sub-layouts within. The header is modeled as a `Row` containing the header text and the body is modeled as a Row again.

5.3 Tagging in CD-Based Web Application Development

We established the foundation of the generic Tagging-DSL in Section 3.5. As mentioned above the generic Tagging-DSL can be used to create a tagging language for a specific DSL in order to enable the adding of *Tags* to that DSL. For this reason, the Tagging-DSL variant *Tagging4CD* was developed. As the methodology presented in Chapter 2 aims at a generic data-centric system, additional options to adapt and customize the outcome need to be provided. Tagging4CD enables the system modeler to add additional information to the class diagram without changing the original CD4A-grammar or any CD4A-model. This is especially useful if the model does not originate from the system modeler himself and, thus might not be modifiable by him directly. In the following, we take a look at the application of the Tagging language to the CD4A language in the context of model-driven web application development:
Tagging requires next to the common languages (L^{TAG}_{Common} and L^{Schema}_{Common}) described above a sublanguage L_G from which the tagging language L^{TAG}_G can be derived. Within this thesis, we will use tagging primarily in model-driven approaches that rely on data-structure models. Thus we will take a closer look at the tagging languages based on the Class diagram DSL *CD4A*: L^{Tag}_{CD4A} and $L^{TagSchema}_{CD4A}$ (*cf.* Figure 5.12).

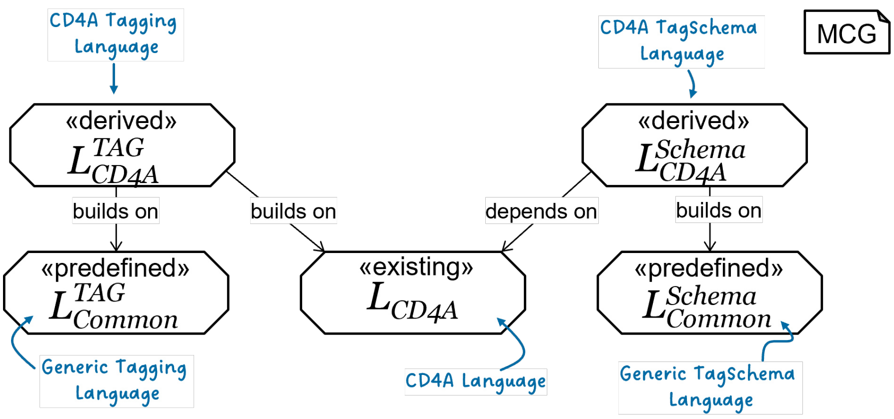


Figure 5.12: Grammar relations of a tagging-DSL for CD4A based on Figure 3.4

5.3.1 Tag Schema for CD4A

The data-structure model can be used to generate major parts of the application. However for some use cases, additional information that is not contained within a typical CD4A model is required. Instead of extending the DSL or extending the generated code, it can be more efficient to tag the information to the CD4A model:

```
package tagschema;                                              TagSchema

tagschema HumanTags {
    tagtype HumanName:String for Attribute, Class;
}
```

Listing 5.39: Tag schema for `HumanName` as used in MontiGem

```
package tagschema;                                              TagSchema

tagschema DBTags {
    tagtype NoCascade for Association;
    tagtype UniqueDBColumn for Attribute;
}
```

Listing 5.40: Tagschema for `DBTags`, defining additional database configurations as used in MontiGem

The example of a Tag Schema shown in Listing 5.39 allows the modeler to add designation for both attributes and classes in the targeted class diagram, by using a `ValuedTagType` with a `String` as an argument. This is especially useful if the model is used to generate a presentation layer (e.g. graphical user interface). The names the modeler chooses often deviate from the names the end user is familiar with. By adding this information, the end user can be presented with the familiar name for the object in question rather than the technical name used by the modeler. Listing 5.40 allows the modeler to add information to attributes and classes of the targeted class diagram to manage how those elements are handled in the persistence layer.

Tags for CD4A

The TagSchema defined in Listing 5.39 is implemented for the CD4A models used in this thesis. It enables the developer to tag a `String` to either a class or an attribute with the `HumanName`-Tag. In the example below the class `Konto` has four attributes that might have names that are not suitable to be displayed to the end user in a GUI, as developers typically use camel case spelling or other conventions such as `is`-prefix to indicate a Boolean parameter. By adding the tag we can provide additional information on how elements in the user interface should be designated. Interfaces and dialogs can be generated, based on these tags and do not have to be refined to be human-readable. Based on Listing 5.39 an Error Message generated because of the missing parameter `sapDatum` could be generated to "Date of SAP-Import is Missing" instead of "sapDatum is Missing".
The Tags used, vary with the use case and target domain, and will be introduces with

the corresponding projects throughout this thesis.

CD4A

```
classdiagram MyDomainModel {
    class Person {
      String name;
      String address;
      Date age;
    }
}
```

Listing 5.41: Example of a simple class. For corresponding class Diagram *cf.* Figure 7.5

Tag

```
package domain; //package of cd model

conforms to tagschema.HumanTags; //TagSchema described in Listing 5.39

tags HumanTags for DomainModel {
    tag Person with HumanName = "Mitarbeiter";
    tag Person.age with HumanName = "Alter";
    tag Person.name with HumanName = "Benutzername";
}
```

Listing 5.42: Tags for the class in Listing 5.41, defining names for the attributes of the class that can be used in user interfaces.

Chapter 6

GUI-Model Derivation from Class Diagrams

In the following, we take a closer look at the model-to-model transformer CD2GUI:

Contents

In order to generate a web-based information system, we need to specify the user interfaces that should be included in the application. However, as we can not anticipate every possible use case, we need to include a mechanism that enables the system modeler to pick and choose from his templates. In the first part of this chapter, we will focus on the model definition of user interfaces for a generic information system. In the second part (Section 6.6, Section 6.7), we will focus on ensuring the flexibility and adaptability of these generic models such that the approach can be applied to a wide variety of use cases.

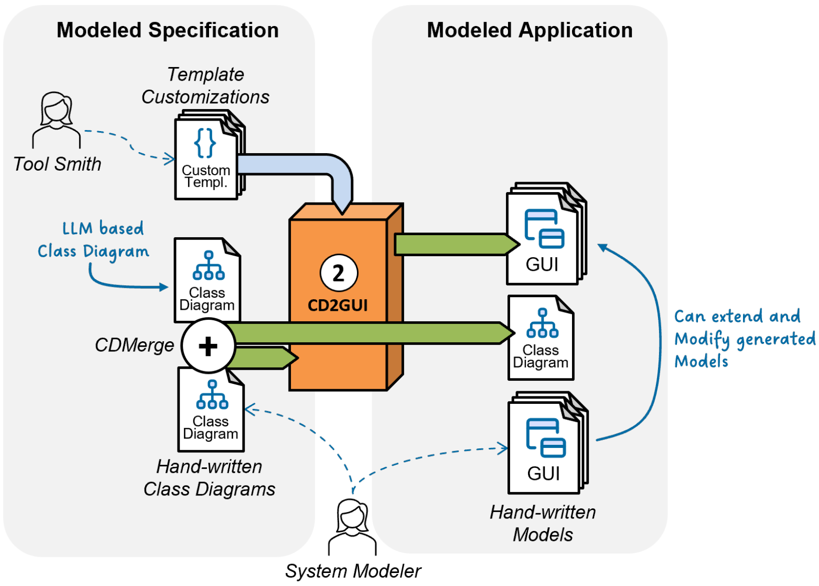


Figure 6.1: Transforming domain models into system models. A system modeler can add hand-written class diagrams to the domain models. CD2GUI derives class diagrams into GUI-models. CD2GUI can be configured with custom templates to add type-specific GUI-model-transformations.

There are several techniques to visualize information and to define corresponding models [BDC23]. We need a transformer that produces these GUI-models based on the given class diagram (*cf.* Figure 6.1) According to Encyclopedia Britannica[1] an information system is defined as follows:

[1]britannica.com

Definition 8. ***Information system****, an integrated set of components for collecting, storing, and processing data and for providing information, knowledge, and digital products.*

The National Institute of Standards and Technology (NIST)[2] defines an Information System as follows:

Definition 9. ***Information system****: A discrete set of information resources organized for the collection, processing, maintenance, use, sharing, dissemination, or disposition of information.*

The Principioa Cybernetica Web[3] provides the following definition:

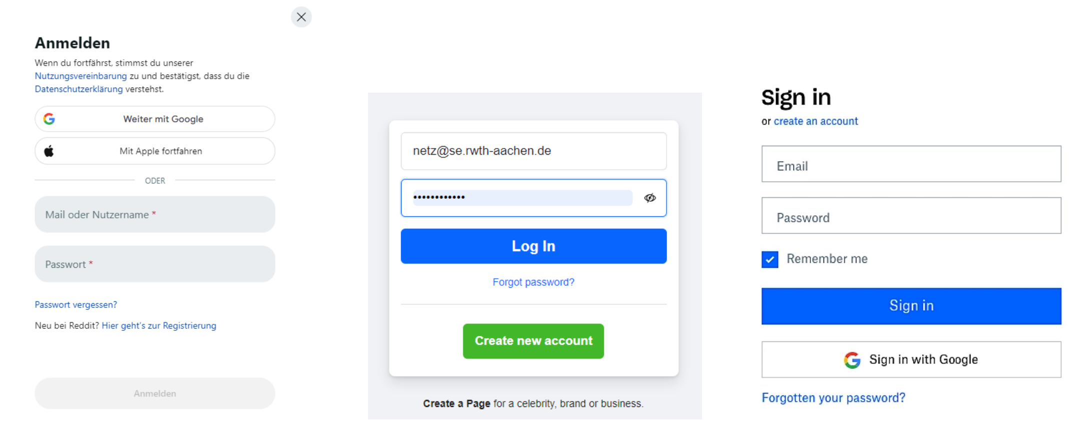
Definition 10. ***Information system****: A system of functions concerning the acquisition and transfer of information, the carriers of which can be biological, personal, social or technical units. An information system is dedicated to a certain kind of information (topic), even if this may be a very broad one. It has always the purpose of providing information to a user or a group of users. In most cases a storage device is part of an information system.*

In order to create such a system a set of domain-model dependent pages is needed in our web application, that allow the user to enter an retrieve data for that model. Next to these pages further pages that needed independently from the model provided:

- **Login**: A simple page ensuring that each user is authenticated and has the necessary privileges to access data of the system. A login is necessary to ensure access restrictions, and in order to prevent general access to potentially private data.
- **User-Management**: A small set of pages providing the user with options to manage their own profile and to recover a password in case it was forgotten. User management ensures flexible granting of access permissions to any registered user.
- **System Pages**: Pages for the imprint, page-not-found page, and generic data privacy regulatory information as well as a generic landing page. These pages contain regulatory information such as information according to general data protection regulations (DSGvO).

The GUI-models for these pages usually do not need to be generated or at least not fully be generated and can be part of a set of predefined static pages, as the basic structure of these pages tends to be the same among all use cases *cf.* Figure 6.2 (e.g. changes in the domain model will not lead to changes in the 404-page).

[2] nist.gov
[3] pespmc1.vub.ac.be



(a) Login Page from Reddit (b) Login Page from Facebook (c) Login Page from Dropbox

Figure 6.2: Login pages share a common design pattern, that is typically independent from any modeled data structure (username, password and often optionally a single sign-on interface such as google).

Next to these pages further pages might be useful to improve the usability and user experience of the application. Some options are:

- **Change Data Capture:** A global log or history of actions throughout the application (*cf.* Section 6.3.1).
- **Navigatable UML** A data overview showing what data is available and how it is organized (*cf.* Section 6.3.2).
- **Search:** A global keyword search that screens the entire database (*cf.* Section 6.3.3).

There are many options to extend the generator in order to cover more use cases. For this reason, we keep adaptability (*cf.* Section 6.7) and variability (*cf.* Section 6.6) of the generator in mind while developing the approach.

6.1 Research Method

The development of the CD2GUI transformer follows the case study approach. A methodology is developed for a specific use case and is generalized later on. The initial scope of CD2GUI is the development of user interfaces for data-centric information systems in the finance sector. Therefore, we need to define an interface that an end user can analyze and manipulate the data structure defined by the previously synthesized domain model. We collect data on both the usage of these interfaces and the software engineering processes related to the produced user interface models in multiple use cases

[CMNR24, MNN+22]. The data is analyzed in order to uncover deeper insights, and find potentials for improvements of the transformer. Thus updating and refining the used languages (GUIDSL v1 → GUIDSL v2) and the corresponding generators.

6.2 Schneidermann's Mantra: Some Core Pages

As defined above information systems are intended to provide the user with access to information. In order to do so we follow the principles of the Schneiderman Mantra [Shn96, CCOTF09]. This mantra encapsulates a design philosophy that prioritizes user experience in interacting with information systems, allowing them to start with a broad perspective and incrementally dive deeper into the details as required, thus facilitating a more intuitive and effective exploration of the data [CC05]. Schneidermann proposes the following key actions in order to create an effective data visualization:

- **Overview**: Gain an overview of the entire collection.
- **Zoom** : Zoom in on items of interest.
- **Filter**: filter out uninteresting items.
- **Details-on-demand**: Select an item or group and get.
- **Relate**: View relations hips among items.
- **History**: Keep a history of actions to support undo.
- **Extract**: Allow extraction of sub-collections and of the details when needed.

As our approach is data-centric and as our single underlying model is a class diagram we can derive the following pages that we need to synthesize, based on Schneidermann. We can implement the actions with the following kinds of pages:

6.2.1 Dashboard Page

A generic page (*cf.* Figure 6.3), that gives an overview of all data currently stored in the system. This page targets primarily Schneidermann's first mantra: *Overview*. It serves as a starting point to navigate to more specific pages and provides meta-information about the stored data. The dashboard page serves two purposes. First, provide the user with an overview of the data currently stored in the database. Second, a means to directly navigate to any of the overview pages of the modeled classes.
The generated dashboard can be modified and extended to show use case-specific visualizations, such as pie charts and aggregated data sets.
Figure 6.4 depicts the class diagram for both Person-Class and Employee-Class that are used in the following examples.

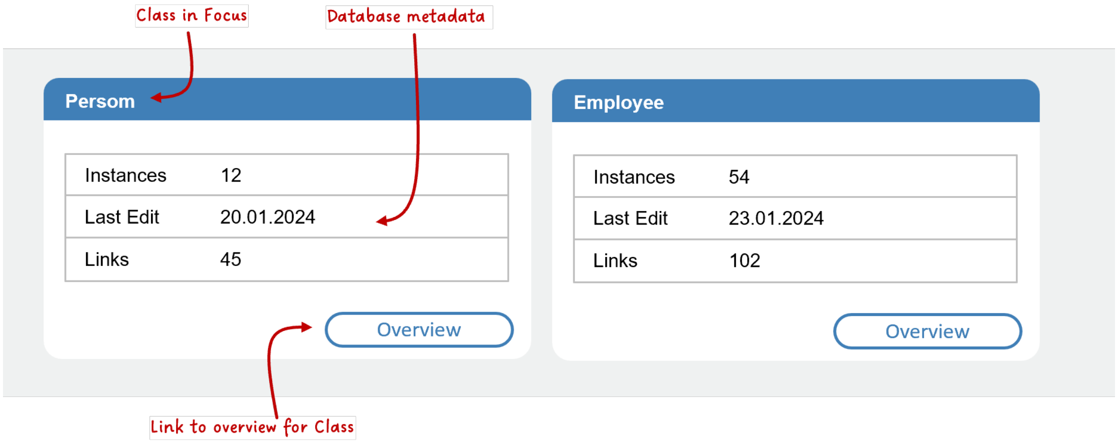


Figure 6.3: Generated Dashboard page for Person an Employee Class.

Similar to template hook points [HKR21] the dashboard can be used as a generic placeholder for a landing page or further generic pages, that do not relate to one specific type. The generated page or parts of it can be overwritten by custom hand-written models.

6.2.2 Class-Overview Page

A page that is created for each class of the root class diagram. The overview page (*cf.* Figure 6.5) aims at the *zoom* and *filter* actions of Schneidermann's mantra. Coming from either a navigation bar or the dashboard, the user can 'zoom in' on objects of a specific type. The overview page provides a list of all elements. The table is provided with options to filter and search entries, thus helping the user to keep an overview. The table components that we use however allow grouping and export of data thus fulfilling at least in part both the *history* and *extract* mantra.
In addition, the overview page also provides an option to enter new objects into the list. Enabling the user to create a new object of the overview page-specific type.

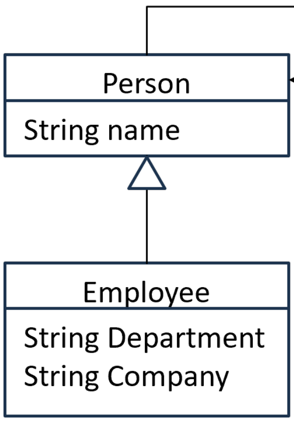


Figure 6.4: Class Diagram used for the examples showing a dashboard page (*cf.* Figure 6.3), an overview page (*cf.* Figure 6.5) and a details page (*cf.* Figure 6.6).

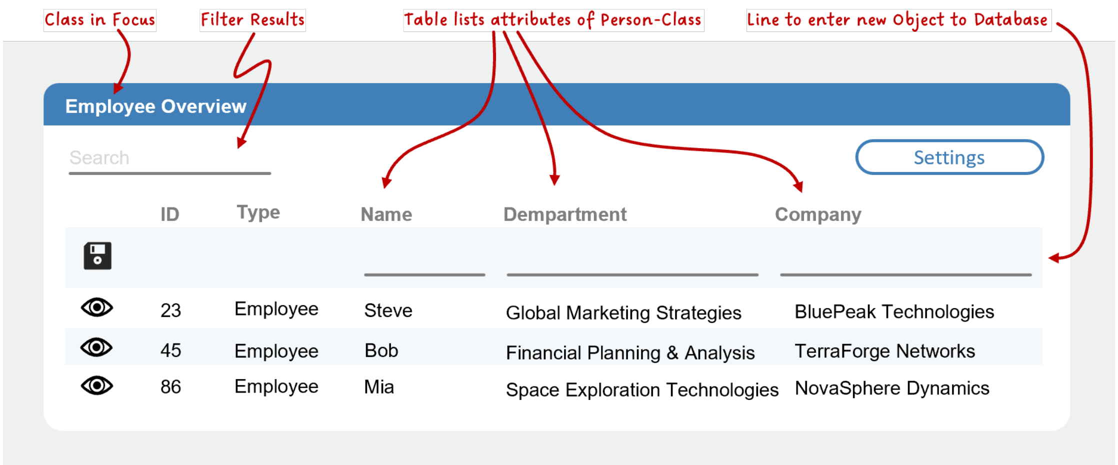


Figure 6.5: CD2GUI default overview page for all Employee-Objects.

Figure 6.5 provides an overview of the page configuration. The overview page primarily consists of a table component, that lists all objects of a specific type and any type that is a subtype. The example shown is the overview page for the *Employee* class. Each entry in the Table is interactive and links to the details page (*cf.* Section 6.2.3) of the corresponding object. Depending on the GUIDSL variant additional features are provided with the table. The table component of GUIDSL v1 (*cf.* Section 5.1.3) provides many features, such as sorting, filtering, and data export. The component provides several extensions, as it was improved over the years of development of MaCoCo. The first line in the table can be used to define a new object. The table component of

GUIDSL v2 only provides basic functions but is very likely to be extended in upcoming projects.

6.2.3 Object-Details Page

A details page (*cf.* Figure 6.6) is created for each class. It is intended to provide information on a single object. The details page can be accessed by clicking on an object listed on the overview page. This action is based on the *details-on-demand* mantra. Next to the information on the object itself, the details page also shows the relations the object has to other objects. The user can inspect and navigate to any other linked object using the details page, following the *relate*-mantra.
The fields shown in the details page can also be used to edit the shown object. In case the user needs to change data he can do so directly, following the 'what you see is what you get' principle. The page shown in Figure 6.6 can be divided into two parts: the upper part provides the attributes of the object in focus. Attributes can be edited directly, either by entering the text directly or by using specialized components such as date pickers or dropdown menus. The lower part of the page provides information about any linked objects. Each association modeled in the input class diagram is represented with its own table, as long as it has a cardinality greater than one. Associations that only allow the link to a single object are treated like an attribute and thus are editable by the upper part of the page. An association-specific table lists all currently linked objects. Every listed object can be unlinked or can be clicked on to navigate to its corresponding details page. Below the list of linked objects is another one: A list of linkable objects. Linking via the 'add' button moves objects to the first list, and clicking on 'remove' moves objects to the second one. All association-specific tables do not only list objects of the association type itself but also handle subtypes as well, similar to the table of the overview page.

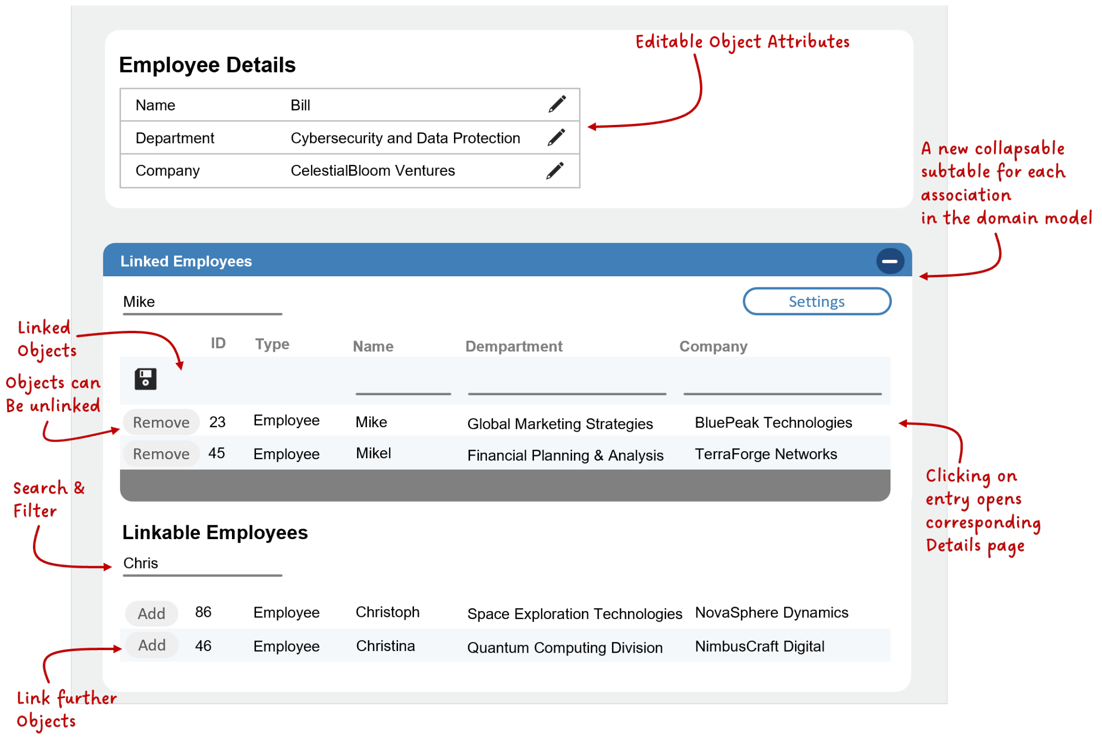


Figure 6.6: CD2GUI default details page an Employee-Object. Inherited attributes (e.g. 'name') are also listed.

In order to enable the user to view and edit specific objects, we need a type-specific page that accesses individual objects from the database. For each class in the root class diagram a details page (*cf.* Figure 6.6) is generated. The details page serves three main functions:

1. **Visualization:** The page presents all attributes and linked objects of a specific object to the user.
2. **Data Modification:** The page enables the user to change any attribute of the object and provides functionalities to link and unlink objects from associated classes.
3. **Navigation:** The page enables the user to navigate to the details page of any linked object. Enabling a data traversal through any related objects.

The details page provides in-depth information on each object of a specific type. The page lists all attributes and associations of the object and provides functionality for editing the object. It also provides links to related objects in order to enable navigation through the entire data structure of the single underlying model.

6.3 Additional Pages for the Web Application

The principle idea of starting from a zoomed-out perspective and incrementally diving deeper into more detailed views of the data can be realized with three kinds of pages: A *dashboard*, an *overview page*, and a *details page*. We aim to provide an application with a high degree of usability (Section 1.4.4), therefore the generator provides additional pages that add additional functions and reduce the developers effort. The following section presents further pages that are generated, based on the single underlying model.

6.3.1 Change Data Capture

Based on the input class diagram, a page can be synthesized that shows a history of all database interactions (*cf.* Figure 6.7) implementing the *history* action.
One of the most important features of an EIS is the provision of real-time data access and reporting [AKJP05, SBLY08]. This helps in better decision making, as managers and leaders can access the most recent data and make decisions based on that information. The features presented up to this point only visualize the data as-is and do not give insight into actions from the past or historic data. As we are generating both the persistence and the communication layer we can extend the generator to add a Change Data Capture-GUI that provides the user with an overview of all past actions and historic data states. In the context of this thesis, we implemented two variants of this feature: (1) an extension of the generated command infrastructure (*cf.* Section 7.3.1) with a memento pattern or (2) an extension of generated data access objects (*cf.* Section 7.3.1) to include the memento pattern. In both cases, the current state of the affected objects is stored before it is changed and can be visualized with a dedicated GUI as shown in Figure 6.7. Since we do not lose the type information on the history records, we can still apply RBAC (see Section 6.5.1) to the history records so that no user can access restricted data. Both approaches are realized by extension of the function library of the generator framework to synthesize additional methods that handle change data capture in either the command class or the DAO class that corresponds to a specific class defined in the base class diagram.

6.3.2 Navigatable UML Class Diagram

Depending on the size of the data structure and the corresponding use case, a simple sidebar navigation might not be optimal to provide the user with access to the data. Therefore, we explored different dashboard variants that list all available classes.
We can visualize the root class diagram by transforming it to PlantUML. The resulting SVG is enhanced to be interactive, so that clicking on a class navigates to its corresponding overview page. As class diagrams can be extensive, the diagram is scalable with a simple slider. The resulting generated component is shown below in Figure 6.8. This variant is defined by a combination of a single GUI-model and a generated SVG

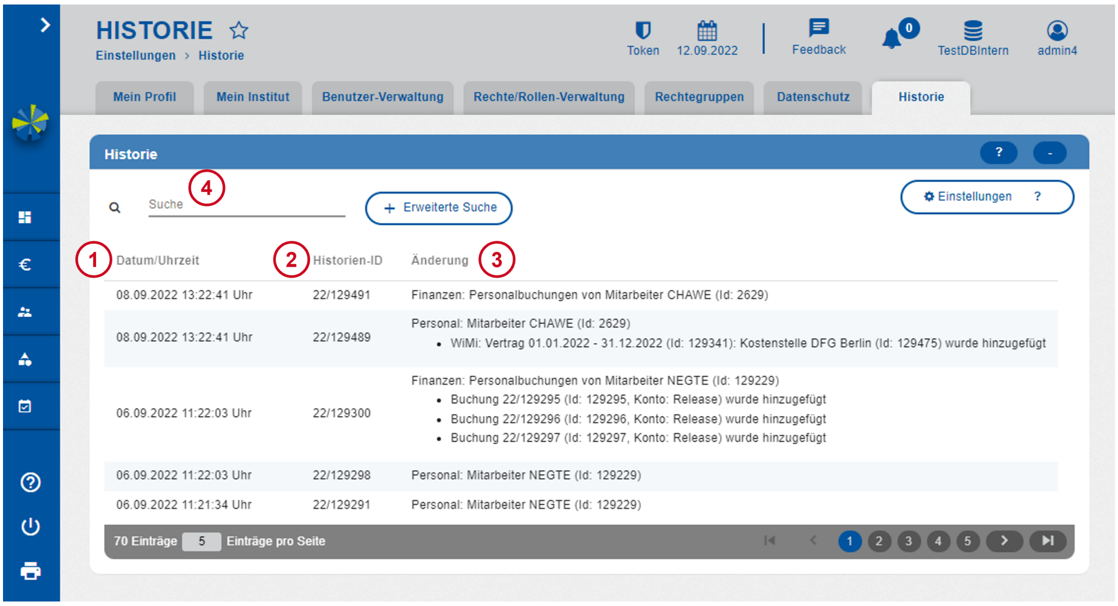


Figure 6.7: MaCoCo screenshot showing the history function of the EIS. The information displayed is (1): When an action was performed, (2) a Unique event-id, and (3) a collapsable description of the action. Additionally, all actions are searchable

file. A drawback of this navigation-variant is its technicality: Although it visualizes the precise relations between all navigable classes, it contains to much irrelevant information for users with no software engineering background. Also, this variant relies on the generated SVG file, which only can be used 'as-is' and cannot be extended with our established mechanisms.

6.3.3 Global Search

Finding relevant Data in an Information system is one of the most important aspects of the platform. Although the navigation previously presented aids in finding specific types, it still requires the user to traverse hierarchical structures to find searched data. Relying only on the input CD we can generate both the user interface and search algorithm, which is optimized for the generated data structure. Similar to the generic approach shown in Section 6.4, another GUI-model for the *Global Search* is generated. This model is similar to the overview page, except that it is not type-specific. Next to the model, the data structure generator is extended to provide an additional search algorithm. In addition to the previously introduced basic functions of the DAO (Section 7.3.1) type-specific search functions are generated. Figure 6.10 depicts the generated search architecture:

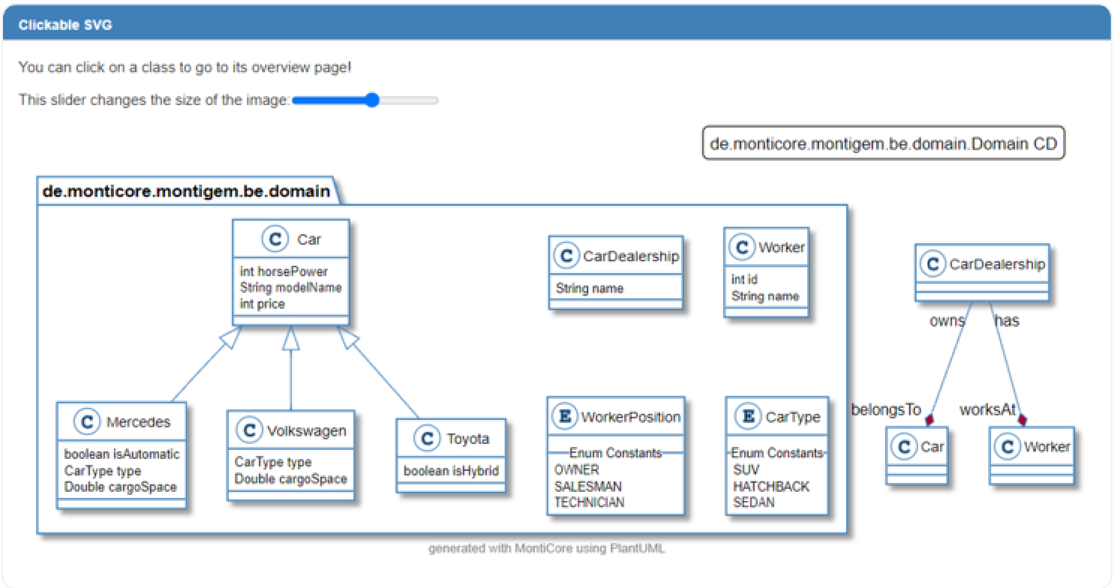


Figure 6.8: Visual Representation of the input Class Diagram used as web page navigation. This page is defined by a combination of GUI-Model and generated SVG File.

Starting at the generated GUI-Model, a generic search command containing a search term is sent from the Client (FE) to the Server (BE). As the command is generic, it does not hold any information on any types or databases it can search, thus it accesses the generated `DAOLib` in order to iterate over all available types. `DAOLib` itself calls search algorithms within the specific DAOs that can access the database directly. Search results are returned as a generic `SearchResult`-Type containing type information, target-Id, and a result string, that indicates both the found element within the object, as well as its surrounding context. The resulting User interface is shown in Figure 6.9: A user can search for keywords and toggle case sensitivity. The results are listed in the table below. Objects of any type are listed in the same table as every object is represented with its type and ID, and its values are represented as a string. Any match with the search keywords is highlighted. The user can navigate to the details page of each object by clicking on it or by clicking on the 'eye-icon' at the start of each entry. The user can also delete elements with the 'bin icon'. We enable the visualization of any results by shortening the object details to show only the part that matches any keyword. In order to get full details on the object the user has to navigate to the details page.

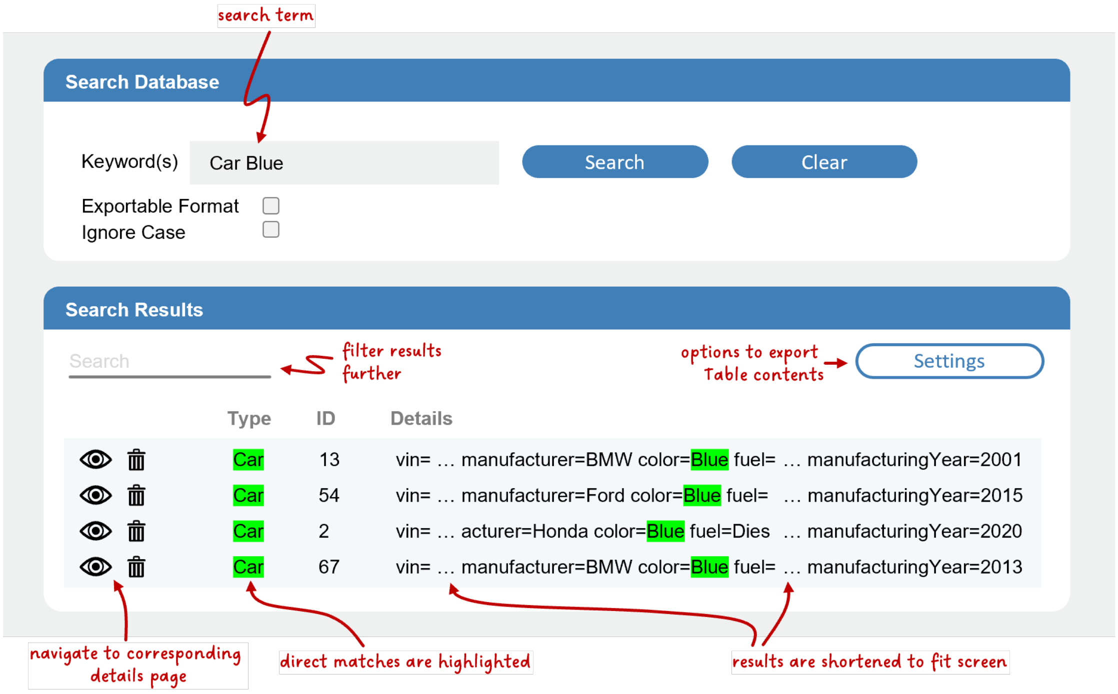


Figure 6.9: The user receives a search interface that can be used to search the entire database. A click upon a search result navigates to the *Details Page* (Section 6.2.3) of the corresponding object. The search can be toggled to be case-sensitive and supports common features such as quoted search terms.

6.4 Architecture

The goal behind the development of CD2GUI is the creation of GUI-models for user interfaces, based on a class diagram in order to provide the user with intuitive and effective access to the data stored in the database. Within the proposed toolchain, CD2GUI is placed after LLM4CD and before MontiGem. It receives the class diagram from LLM4CD creates a set of GUI-models based on it and passes them and the class diagram on to MontiGem.
In order to provide a GUI for efficient data access CD2GUI needs to provide a generic *dashboard page* an *overview page* for each class and a *details page* for each class.
Figure 6.11 gives us an overview of the main architecture of CD2GUI. Starting on the left CD2GUI is provided with both a class diagram and a configuration. In our approach, the class diagram is provided by the preceding transformer: LLM4CD. The configuration is set up in advance by the developer of the toolchain. The configuration primarily contains paths for both the input and output of artifacts and paths to templates that should be

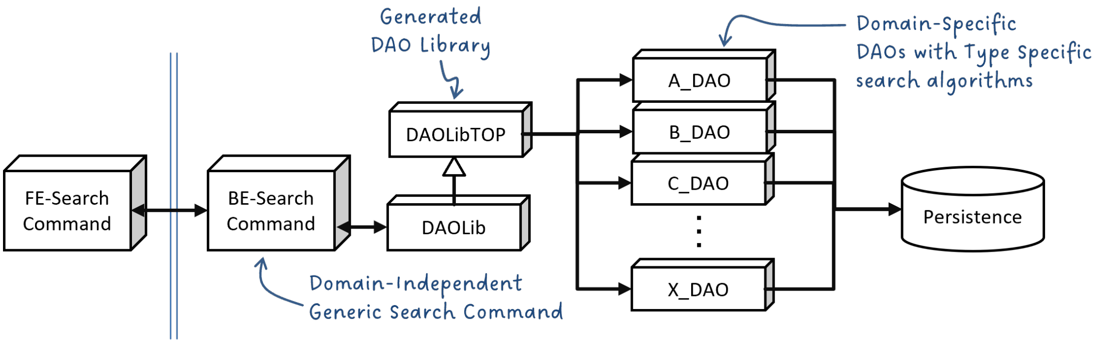


Figure 6.10: Extension of the RTE, and Generator enables the generation of a type independent domain-specific global search through the entire database, that the user has access to.

used. The *model loader* parses the root class diagram and prepares a list of classes that are used by the *GUI model creator*. The GUI model creator provides the generators that create for each kind of page a set of corresponding models. Each generator iterates over the provided list and uses a set of templates to create either a set of pages or, in the case of the dashboard, a single one.

6.4.1 Template Usage

The templates used by CD2GUI are organized in a tree structure, using separate templates to define components and subcomponents of each page. There are four sets of templates: One for each generated kind of GUI-models and generic templates to include new pages in the application. An overview is shown in Table 6.1. The relations between the templates are depicted in Figure 6.12.

```
dashboard-gui.ftl
 ├── imports-dashboard.ftl
 ├── card-dashboard.ftl
 ├── table-header-dashboard.ftl
 └── table-content-dashboard.ftl

detail-gui.ftl
 ├── imports-details.ftl
 ├── attributes-card-details.ftl
 ├── associations-card-details.ftl
 ├── associations-table-details.ftl
 └── details-edit-gui.ftl
```

```
overview-gui.ftl
 ├── imports-overview.ftl
 ├── card-overview.ftl
 ├── table-header-overview.ftl
 ├── table-content-overview.ftl
 └── subclass-overview.ftl

form-gui.ftl
 ├── attributes-card-form.ftl
 └── associations-card-form.ftl
module.ftl
routing.ftl
```

Page	Template name	Details
Dashbard	dashboard-gui.ftl	Defines basic structure of the Dashboard
	imports-dashboard.ftl	Defines imports needed by the dashboard GUI-model
	card-dashboard.ftl	Cards for each type within the dashboard
	table-header-dashboard.ftl	Defines table headers of tables within cards
	table-content-dashboard.ftl	Defines Corresponding content of tables
Overview Page	overview-gui.ftl	Basic structure of Overview page
	imports-overview.ftl	Defines imports needed by the overview GUI-model
	attributes-overview.ftl	Handles type-specific visualization of attributes
	card-overview.ftl	Hanldes card containing overview table
	subclass-overview.ftl	Provides navigation to overview pages of subclasses
	table-header-overview.ftl	Defines Table headers of tables within cards
	table-content-overview.ftl	Defines Corresponding content of tables
Details Page	detail-gui.ftl	Basic structure of Details page
	imports-details.ftl	Defines imports needed by the details GUI-model
	attributes-card-details.ftl	Handles type-specific visualization of attributes
	associations-card-details.ftl	Handles type-specific visualization of associations
	associations-table-details.ftl	Handles type-specific list of associations
	details-edit-gui.ftl	GUI for editing of data of a given object
Form Page[1]	form-gui.ftl	Basic structure of Form page
	attributes-card-form.ftl	Handles type-specific visualization of attributes
	associations-card-form.ftl	Handles type-specific visualization of associations
Generic	module.ftl	Includes new pages in generated application
	routing.ftl	Sets routing for new pages in application

1: Form Page is a variant of the Details Page intended to create new objects, in contrast to the default Details Page, the Form Page does not yet reference an existing object from the database.

Table 6.1: Primary templates used to generate an information system using GUI-DSL v2. In comparison to the approach used with GUI-DSLv1 (*cf.* Table 6.2), the templates are divided in further segments in order to ease the use of template replacement (*cf.* Section 6.6.1).

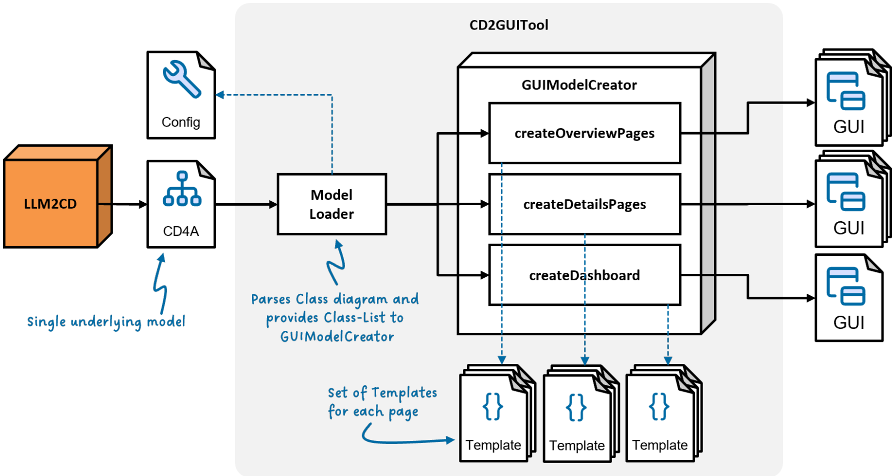


Figure 6.11: Overview on CD2GUI as an extension within the MontiGem-framework

Figure 6.12: Template nesting of templates used in CD2GUI

Each template is used to assemble a GUI-model according to the properties of the provided class. We have to tackle for each template the problem of how to provide a generic method to visualize non-primitive data types. Equivalent to a 'toString()' function, CD2GUI has to provide a solution on how to present arbitrary data on the user interface.

We take a closer look at the templates used to create a GUI-model for the *details page*: Listing 6.1 depicts the basic structure of a Freemarker template to create a GUI-model in GUIDSL v2 syntax. Below Listing 6.2 depicts the corresponding section of the generated GUI-model. The template is called for a single class, as the GUI model creator iterates over the classes provided by the model loader. The template is provided with the class name, its package, attributes, and associations (Line 1). Next, the components and data classes that are used in the GUI-model are imported (Line 2). Each GUI-model that defines a page starts with the `page` keyword. Each model is named after the corresponding class e.g. *PersonDetails*. The page component itself is declared with the object in question for that derails page: `page personDetails(Person person)` (Listing 6.2 Line 2). Within the page component, we can define further GUI components. Each component is defined with a unique name within the GUI-model. We can rely on the naming conventions of class diagrams to provide unique class and attribute names to name each component within the created GUI model. E.g. we can assume that there are no two attributes within one class that share a name, thus we can name components by the attributes, classes, or associations they handle. As shown previously in the schematic representation of a details page (*cf.* Figure 6.6) the GUI-model contains

Page	Template name	Details
Dashbard	classes.ftl	Complete GUI-model for dashboard
Overview Page	overview-component.ftl	Complete GUI-model
	overview-component-ts.ftl	Typescript logic for validators
Details Page	detail-component.ftl	Complete GUI-model
	detail-component-ts.ftl	Typescript logic for validators
	imports.ftl	Imports for TypeScript classes
	component.ftl	Typescript class structure
	constructor.ftl	Constructor for details page component class
	role_methods.ftl	Typescript logic for permission management
	edit_save_method.ftl	Typescript logic for data persistence
	derived_attributes.ftl	Typescript logic for derived attributes
Generic	module.ftl	Includes new pages in generated application
	routing.ftl	Sets routing for new pages in application

Table 6.2: Templates used to generate an information system using GUI-DSL v1. As GUIDSL v2 supports the nesting of models, we are also able to nest templates to a higher degree.

multiple cards. Listing 6.1 Line 4 shows the definition of the first of these cards resulting in `employeeDetailsCard@GemCard` (Listing 6.2). This particular card is intended to list all attributes that are defined within the given class (Listing 6.1 Line 7)

```
${tc.signature("name", "domainPackage","attributes","roles")}
// imports and static parameters ...
page ${name}Details(${name} ${name?lower_case}) {
    ${name?uncap_first}DetailsCard@GemCard(
        title = "${name} Details",
        component = ${name?uncap_first}DetailsRow@GemRow(components = [
// list of attributes ...
        ])
// further components ...
}
```
FTL

Listing 6.1: Basic structure of a Freemarker template to create a GUI model in GUIDSL v2 for a details page.

```
// ...
page employeeDetails(Employee employee) {
    employeeDetailsCard@GemCard(
        title = "employee Details",
        component = employeeDetailsRow@GemRow(components = [
           // some components ...
        ])
    );
// ...
}
```
GUI-DSL v2

Listing 6.2: Corresponding section of the GUI-model for Listing 6.1

The method of data visualization depends on the type of data that should be shown in the user interface. For this reason, we have included a case differentiation in our templates, which visualizes the data differently depending on the type handled. For the details page, there are four distinctions as shown in Listing 6.3. Booleans (Line 3) are displayed as checkboxes (`@GemCheckBoxInput()`) and primitive types (Line 5) such as String or Integer are shown directly as a text field (`GemText()`). A nonprimitive type can not be displayed as easily as primitive ones, as they can have an arbitrary number of attributes themselves, and the class diagram does not provide information on which of those would be helpful to the user to identify the attribute. For example, An employee can have a 'car' attribute that itself has a'modelname' and a 'numberplate' attribute. However, it is unclear, based on the class diagram, whether the numberplate or the model name should be used as an identifying element, to show which car is an attribute of the employee class. Circumvent this issue by providing a link to the specific attribute. Enabling the user to investigate the attribute further. Line 7 shows the corresponding GUI Component `@GemLink()`. Finally, in case the attribute is neither primitive nor has a detail page, a simple text (line 10: `@GemText()`) is shown notifying the user that this element cannot be handled. A handwritten extension is recommended for these edge cases.

```
// ...
<#list attributes as a>
    <#if a.isBoolean()>
        ${name?uncap_first}_${a.getName()}@GemCheckBoxInput(labelText = "${a.getName()}"),
    <#elseif a.isPrimitive()>
        ${name?uncap_first}_${a.getName()}@GemText(value = "${a.getName()} = " + ${name?lower_case}.${a.getName()}),
    <#elseif a.isClassWithDetailsPage()>
        ${name?uncap_first}_${a.getName()}@GemLink(url = "/cd2gui/models/${a.getType().getName()?cap_first}Details/" + ${name?lower_case}.${a.getName()}.gemId, text = "${a.getName()}"),
    <#else >
        ${name?uncap_first}_${a.getName()}@GemText(value = "No Details Page Exists"),
    </#if>
</#list>
// ...
```

Listing 6.3: Template defining a list of attributes (Listing 6.1 Line 7)

This concludes the first card of the details page: It lists all attributes of the shown object and provides a visualization of the data for each type it can handle, otherwise, it provides an option to navigate to the details page of the corresponding attribute. The details page contains further cards that contain lists showing all objects that are linked for each association of the given object. Similar to the attributes card the association cards visualize data in case it is possible otherwise they provide links to investigate the linked objects further. The complete template is shown in the appendix: Listing A.3. The same mechanisms to handle nonprimitive data types were applied to the templates for both dashboard and overview pages.

6.4.2 Handling Inheritance

The GUI-models created by CD2GUI, are customized for specific classes as they are defined in the input class diagram. Therefore, for a detailed list of attributes of a class, we do not only take into account the attributes and associations that are defined directly within the class, but also all attributes and associations that are inherited from super classes of that class. We do not show any attributes of potential subclasses. This affects both the generated list of attributes as well as the table for linked objects: The list of attributes is simply extended by the inherited ones. The Table of linked objects can list objects of any type that extend the type of the association. Figure 6.13 depicts a simple class diagram for a use case that involves inheritance and a corresponding object diagram. Using that class diagram as input for CD2GUI would yield three details pages

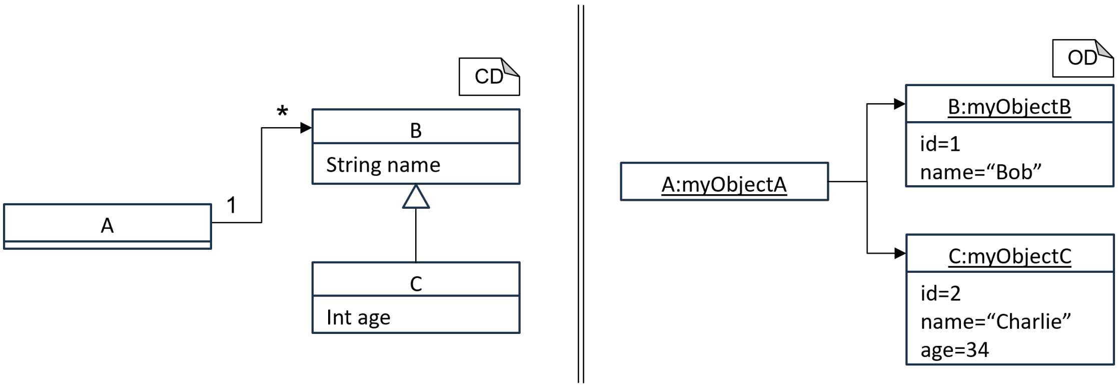


Figure 6.13: Example of inheritance

(for the classes A,B,C), three overview pages, and a dashboard. The details page for class A contains a list of all linked objects of type B. Due to inheritance, objects of type A can also be linked to C and thus also appear in the list. As the list is defined for the Type B, it will only show attributes of the B class. Thus, following the example, the list would show: *myObjectB*, its id and its name, as well as *myObjectC* its id and only its name. The *age* attribute however will not be shown. Clicking on each object will still navigate the user to the correct type of each class.

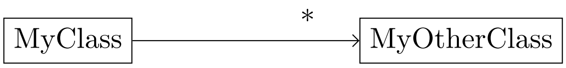
6.4.3 Handling Abstract classes

Abstract classes can not be instantiated; thus we can only list objects that inherit from that abstract type and only provide functions that allow the creation of a new object of a subtype. Therefore, there are still use cases for which a dedicated page for an abstract class can be useful:
Overview page for abstract classes: The overview page lists not only the objects of the type in question, but also all subtypes. Thus, an overview page for an abstract class can serve as a good page to get a generic overview of a subset of classes. In contrast, no user interface is generated for enumerations or interfaces, as they are both abstract and cannot be extended. Enumerations can still be used as a class attribute and will be implemented with GUI components such as drop-downs for editing. Note that any limitations imposed by the default configuration of CD2GUI can be overwritten and extended with handwritten models.
Abstract classes in Details Pages: Details pages for abstract classes can exist for objects of a subtype, providing a view of a subtype as the abstract type. Association to abstract objects can be visualized on the details page in a similar mannor. As it shows, linked object to abstract classes, in case that object inherits from that abstract class. Although no abstract object can be listed, objects of the subtypes will appear in case they are linked

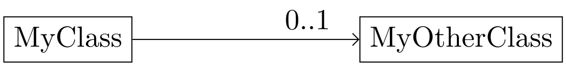
to the object in the focus of the details page.
Abstract classes in the dashboard: The dashboard handles abstract classes as normal classes and provides metadata if available. It also provides navigation options to the overview pages of the abstract classes.

6.4.4 Handling Associations

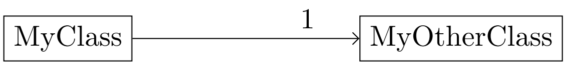
As shown above, we can use CD2GUI to create pages that visualize any type of association. However, the creation of a new object together with linked objects poses a challenge. In the following let us assume we want to create an object of the class 'MyClass'. A class modeled in CD4A can have associations that we have to distinguish by their cardinalities: First, let us take a look at unidirectional associations.



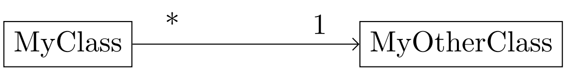
The pages generated by CD2GUI to manage and view objects of type 'MyClass' offer the user the option to pick an existing object of type 'MyOtherClass' to link it to the object in question. If there are already objects linked, the link can be removed and further existing MyOtherClass-Objects can be added. A 'MyClass' object can be created without any linked object.



Similarly to the previous association the pages generated by CD2GUI allow to add, edit, and remove linked objects. In this case, however, the amount of linkable objects of type B is limited to one. CD2GUI will provide the same user interface, but prevent the user from linking more objects.



In contrast to previous associations, the `[1] -> [1]` association requires the linked object B to exist before A is created. Forcing the user to create objects in a pre-defined order. This condition might prevent the creation of objects in case there are cycles in the class diagram. We can tackle this challenge by setting up a transitive closure around any objects that should be linked and generating a corresponding GUI that creates all required objects at the same time. CD2GUI tackles this challenge by generating additional dialogs that allow the creation of the needed object Figure 6.15.
When creating an object we need to take into account both sides of the association:



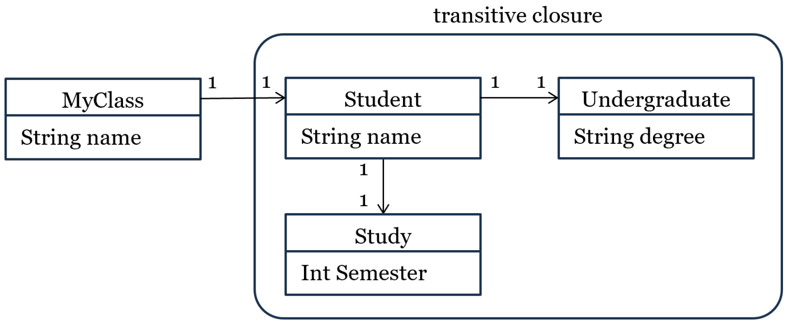
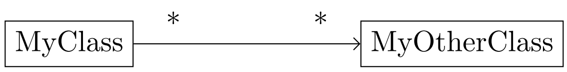
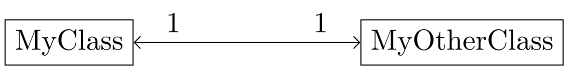


Figure 6.14: Transitive Closure. Corresponding dialog is shown in Figure 6.15

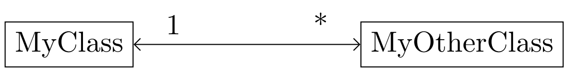
In case multiple Objects of type 'MyClass' can be linked to a single Object of type 'MyOtherClass', the user interface behaves similarly to previous cases, offering the user to set one object of type B as a target. Data validation at both the server and the client ensures that a 'MyClass' object is only linked to one 'MyOtherClass' object.



In case any amount of object 'MyClass' is linked to an arbitrary amount of objects of type 'MyOtherClass'. CD2GUI provides offers to link new and unlink associated objects without any restrictions. Bi-directional associations are treated as the two associations with opposite directions. Thus, the following associations are not supported by CD2GUI:



In this case, neither a 'MyClass' object nor a 'MyOtherClass' object can exist without the other. CD2GUI is required to provide pages to create both objects at the same time. In it's current form there is no user interface to create both the 'MyClass' object and the 'MyOtherClass' object at the same time. We tested an extension to CD2GUI that generates a dialog to create the missing object, allowing to CD2GUI to create both objects at the same time. We solve this issue as described above by forming a transitive closure around all objects that need to be created at the same time. A corresponding GUI is generated to enable the user to set up all objects simultaneously.



This association produces a similar problem as the association described before. The 'MyOtherClass' object can not exist without a 'MyClass' object present and thus has to

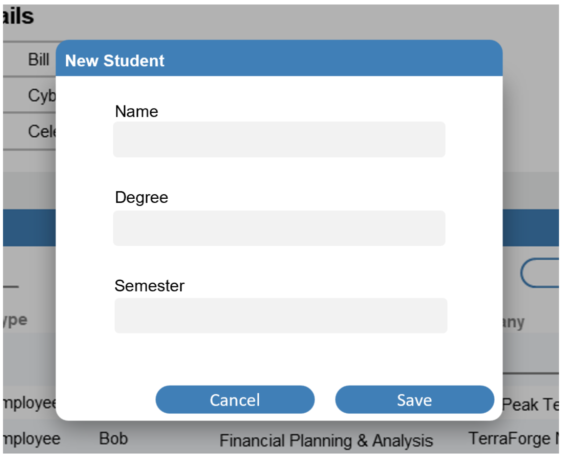
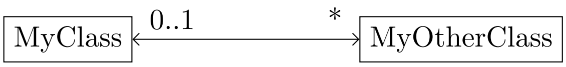


Figure 6.15: CD2GUI tackles transitive closure with a dialog, listing all needed attributes. This enables the user to define any object that is linked to the new object in focus. There is one dialog for each outgoing association of the new object that is currently being edited.

be created at the same time as the 'MyClass' object.



In this case, the 'MyOtherClass' object can exist without a 'MyClass' object linked to it thus we can create both objects in sequence.

6.4.5 Handling Generics

CD2GUI supports the following generics: `Optional<A>` and `List<A>`. `Map<A,B>` and `Set<A>` are not supported. CD2GUI treats lists as associations providing GUIs that offer the same mechanisms to add and remove elements to an object. An optional parameter is also treated like an association with a 0..1 cardinality. Permitting the creation of an object while leaving that specific attribute empty.
Table 6.3 summarizes which elements of a CD4A model are supported by the class diagram model to model transformer. As CD4A does not support packages, generics, and annotations, CD2GUI that is based on the CD4A can not process these elements as well.

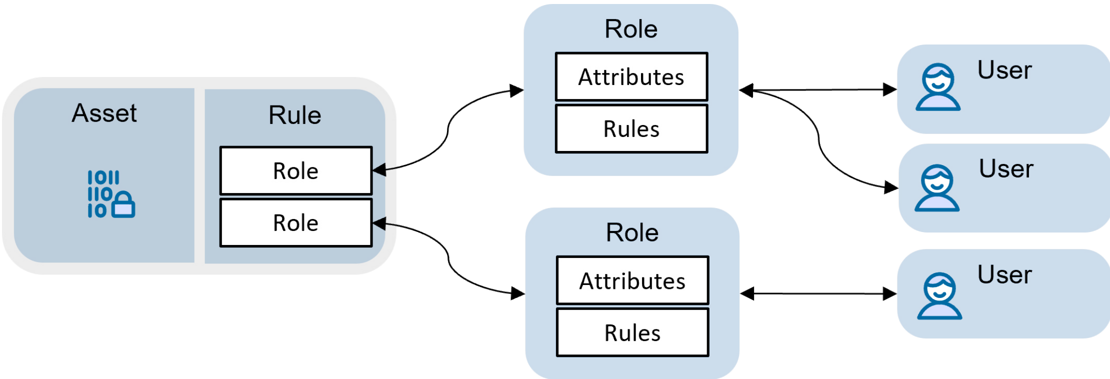


Figure 6.16: Role Based Access Control as Defined in [FCK+95, SFK+00]

6.5 Further Artifacts Derived from Single Underlying Model

CD2GUI targets the creation of GUI-models based on the list of pages we identified in Section 6.2 in order to provide efficient access and manipulation facilities to data. Next to GUI-models we can create further code based on the single underlying model in order to manage data access and finally testing of the system.

6.5.1 Generated Role Based Access Control

An information system is intended to serve multiple groups of users that each might have varying access to the data stored within the system.
Therefore, we need to establish user-specific *access policies* [SS94] to restrict access to the system. The system we propose has been successfully used in MaCoCo (Chapter 8) and is based on *Role Based Access Control (RBAC)* [USMDAS14, FCK+95].
Role-based access control (RBAC) assigns system access to users based on their roles within an organization, each with specific privileges, rather than assigning access on a case-by-case basis as in rule-based systems. Unlike discretionary models where individuals control access, RBAC centralizes access management by aligning it with the roles' defined functions and responsibilities. Figure 6.16 depicts the principles behind RBAC: Multiple *Users* can be assigned to a *Role*. The roles themselves contain privileges and rules. In order to access an asset, a user must be assigned with a corresponding role.
The data structure of the generated application (*cf.* Figure 6.17) is structured as follows: A domain user can be granted access policies. An access policy can contain a unique asset ID stored in the database. It also links to one role. A role itself contains multiple operations (permission) that define an operation that is performed on the asset (`CREATE`, `UPDATE ...`). A role also links to a domain. A domain can group classes within the single underlying model. Any class can be assigned to one domain. The rule to access an asset

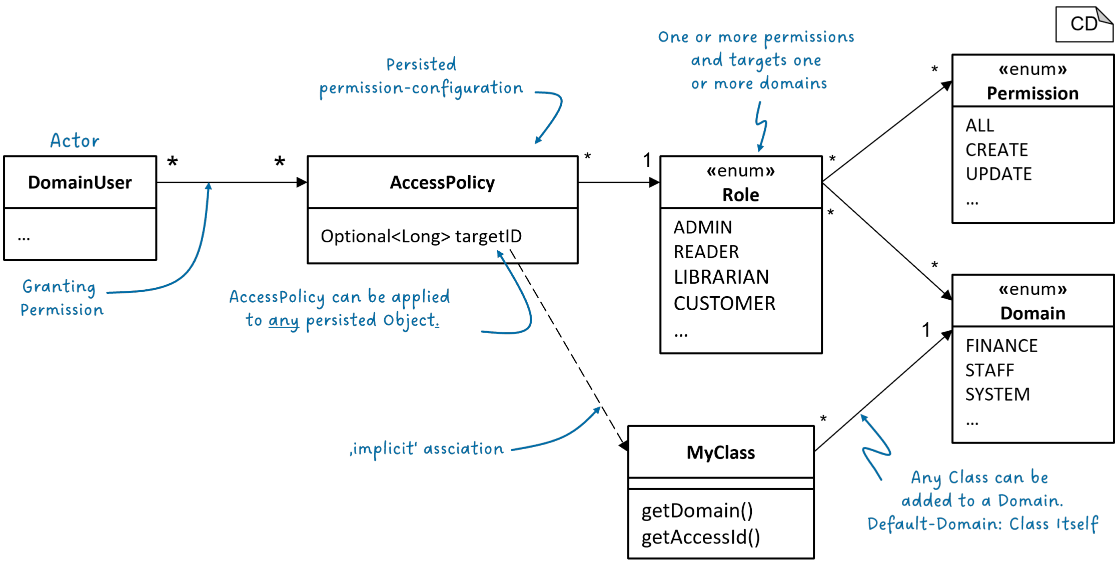


Figure 6.17: Role Based Access Control as defined for a system configured by CD2GUI.

is implemented as follows:

$$User \times Role \times Operation \times Domain$$

A user has a role that defines an operation upon a domain. A user can only access or modify data in the event that he is assigned a role that contains the operation he wants to perform and in the event that the targeted object is in the same domain as defined in that role.
Let us assume that the generated web application is a web store. Roles that would be likely are *Customer* and *StoreOwner*. Setting up the system, the developer would assign the Operation *UPDATE* on the *Price*-Objects to the *StoreOwner* Role. On the other hand, the developer would assign the Operation *READ* on the *Price*-Objects to the *Customer* Role. During runtime, when a new User is created it can be decided what role to assign. The shopkeeper would be granted the *StoreOwner* Role, granting him privileges to change any of the prices in his store. We can restrict the role assignments by mapping a role to a specific subject, e.g. letting a user only see one specific *Price*-Object. RBAC relies on a mapping of roles, operations, and objects that cannot be directly derived from the same class diagram that defines the basic domain of the application. One option to generate such a system would be to create a superset of all CRUD operations and classes and transform them into roles that can be assigned to the users as needed. This shifts the workload from the developer to the end user, as he has to figure out the correct roles that are needed for his day-to-day usage of the web application. This also bears the risk of inconsistent permissions among users which tends to result in

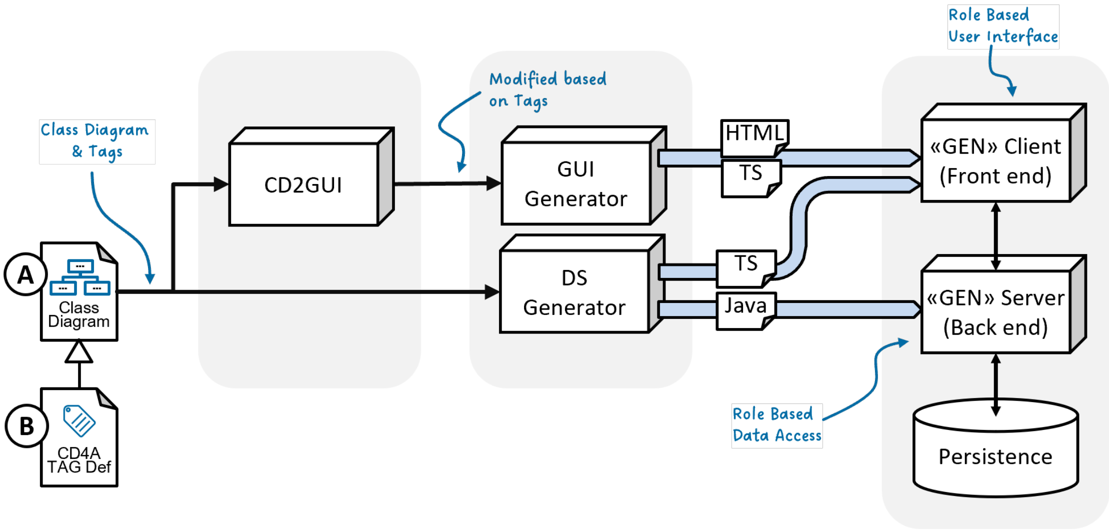


Figure 6.18: Generated Role Based Access Control integrated into MontiGem architecture (*cf.* Figure 6.11)

more rights being granted than necessary. A more efficient solution is the usage of the tagging language (see Section 3.5) to map roles to specific classes and thus have a use case-specific set of roles similar to those mentioned in the web store example above.
Thus we can tag the root class diagram and generate RBAC that is linked directly to the single underlying model.
Figure 6.18 shows the model-driven approach using tagging (*cf.* Section 3.5) in order to define access control. The class diagram can be tagged with role definitions. The tag schema defines how the class diagram can be tagged. Based on the tag schema the tagging generator provides tag-specific methods to handle tagged elements of the class diagram. Both CD2GUI and MontiGem have to be extended with methods provided by the tagging generator to process the tagged CD4A model. Resulting in a generated architecture that implements consistent access management for both the server as well as the client.
In order to reduce to complexity of roles and permissions, every class is generated with the methods `getDomain()` and `getAccessId()` (*cf.* Figure 6.17). The method `getDomain()` is used to return the domain a class is grouped by. Multiple classes can return the same domain. Enabling roles that grant access to a single domain rather than a set of classes. The method `getAccessId()` can refer to either to the ID of the object itself or, refer to the ID of another object. This allows to derive permissions from other objects. For example, if a user has access to a 'Budget' object and there are several 'Bookings' within the budget. By referring to the ID of the Budget we can enable a user to automatically have access to any related 'Booking' in case he has also access to the corresponding

'Budget'. This reduces the amount of assets we need to target in our permissions to key objects in the data structure.

Tag Definition for SUM-Based RBAC

The fundamental concept of access control by tagging is to define a tag type that can then be used in tagging definitions to tag elements of a class diagram model with access permissions. CD2GUI uses a final tagging schema to define tag types for RBAC-related tags. The tagging scheme has been specifically designed for access control, as demonstrated in Listing 6.4, and is exclusively intended for access control purposes rather than any other tag types the generator might employ. This has the advantage of clearly separating the functional logic, allowing the tagging schema to be easily reused for other generators requiring access control. As a result, the tagging schema is referred to as the AccessControl tagschema.

```
package de.monticore.taggingschema;                                TagSchema

tagschema AccessControl {

    tagtype AccessRight: {
      flag: String,
      operation: [R | CR | RU | RD | CRU | CRD | RUD | CRUD | NONE]
    };

}
```

Listing 6.4: Used tagschema for access control. The used tag definition consists of two inner tag types: `flag` and `operation`

The tagtype `AccessRight` used in the tagshema is a complex tag type (*cf.* Section 3.5). The tagtype combines the two tags: `flag` in order to tag a Role to a class diagram element and `permission` to tag permissions to the same element. Thus we can use the combined tag to define what *Role* is needed for which *Operation* if access is requested for a specific element in the class diagram. We take a closer look at the individual functions of both tags:

Tagging a Role: As the roles are defined as a Java enum (*cf.* Figure 6.17), we can not reference them directly from the tagging. Therefore, a value tag type with the type String is used to reference the permission flags in a tag. As can be seen in Listing 6.5, the designed tag type has the name Flag and the type String.

TagSchema
```
tagtype Flag: String;
```

Listing 6.5: Tagtype `Flag`

Listing 6.6 depicts how to tag the class 'Employee' with a Role. The class 'Employee' is tagged with the 'ADMIN' role. Configuring the targeted web application to only grant users with the 'ADMIN' role access to 'Employee'-objects.

Tag
```
tag Employee with Flag = "ADMIN";
```

Listing 6.6: Definition `Operation` Tag.

Tagging an Operation Next to defining what role can access an element, we define what operation can be performed on that element. A role might have only grant a user limited operations for a specific element therefore we need to specify what restrictions apply to what role. For example: An 'ADMIN' might have full access to create, read, update, and delete any Employee object. A 'HR-Assistant'-Role however, only might have reading access to any Employee-Object. It is crucial to differentiate whether a user may only view data records or can also edit, create, and delete them as necessary. The web interface functions can be categorized into CRUD (create, read, update, delete) operations. To address this, a tag type Operation assigns CRUD operations to the elements in the class diagram model.

TagSchema
```
1 tagtype Operation: [CREATE, READ, UPDATE, DELTE];
```

Listing 6.7: Tagging the Employee with `Operation` Tag. The depicted version is the verbose variant. The current implementation uses the shortened variant *cf.* Listing 6.9.

Listing 6.8 shows two tags for the 'Employee' class. To access an 'Employee' object a user has to be assigned a Role that contains either 'Read' or 'Update' permissions.

Tag
```
tag Employee with Operation = [READ];
tag Employee with Operation = [UPDATE];
```

Listing 6.8: Verbose definition of an `Operation` Tag

In this example, the Employee class has been tagged with the initial variant of the Operation tag type and given the READ and UPDATE values. The enum tag type is designed to have only one assigned value in a tagging definition, which means that if multiple operations are to be tagged to an element, multiple tags must be defined as well.

If an item requires all four CRUD operations to be tagged, then four separate tags are necessary. To simplify this process, the Operation tag type has been shortened to allow managing multiple CRUD operations with a single tag. The currently used definition of the tag type is displayed in Listing 6.9.

TagSchema

```
// Shortened variant
tagtype Operation:[R| CR | RU | RD | CRU | CRD | RUD | CRUD | NONE];
```

Listing 6.9: Tagshema for shorter permission definition.

The AccessRight tag type is a complex one that includes both the Flag and Operation tag types. They are defined as inner tag types within AccessRight. Listing 6.10 illustrates the tag type's application. Rather than defining two individual tags for the Employee class, we tag the Employee class with the AccessRight tag. The tag specifies the values 'ADMIN' for the Flag inner tag type and 'RU' for the Operation inner tag type.

TagSchema

```
tagtype AccessRight: {
 flag: String,
 operation: [R | CR | RU | RD | CRU | CRD | RUD | CRUD | NONE]
};
```

Listing 6.10: Complex tag type used in current architecture, grouping multiple permission options within one tag.

Using the complex AccessRight tag type allows for defining multiple tags for the same element, each clearly assigned to a specific Rle and corresponding Operations.

Tag

```
// Only HR can read Employee Data:
tag Employee with AccessRight = { flag="HR-Assistant", operation=[RU] };
```

Listing 6.11: Tag defining what Role is needed with which rights to access an Employee Object: A user needs the 'HR-Assistant' Role to either read or update an Employee object.

Tag-Based GUI-model generation

The RBAC information added to the SUM is used by CD2GUI to create corresponding GUI-models. Both GUIDSL v1 and GUIDSL v2 permit the definition of guards that define if a GUI component is displayed or not. Listing 6.11 gives an example of the guard `ifPermission`, used within a GUI-model to ensure that a button is only shown in case the current user is assigned the 'ADMIN' role.

```
button ifPermission(ADMIN) 'Employees' {                    GUI-DSL v1
    click -> navigateToEmployees() {
    ''this._router.navigate(['cd2gui/employees'],);'' );
}
```

Listing 6.12: GUI-model using `ifPermission` to define an RBAC-constrained user interface.

Depending on the tagging model, guards are applied to all tagged elements of the SUM defining a set of user interfaces that show or hide elements based on the Roles of the signed-in user.
In addition, the tagged class diagram is processed by the generator that provides the infrastructure between the server and the client. Preventing not only the visualization of GUI components but also the transmission of restricted data to the client based on the privileges of the signed-in user.

6.5.2 Generated Test Data

A key element of prototyping is testing. Similar to RBAC, the single underlying model can be used to generate test data. Within the different use cases this generative approach was applied to, we developed four different ways to produce test data.

1. **Random Data:** A simple random number generator (RNG) is used to produce random values during object creation. Although this method is very efficient and resource-saving, it very often does not create data that fits any use cases. It is best suited in unit testing to identify edge cases for specific parameter settings. However, user interfaces require further refined data in order to be optimized to better visualize data of a given use case.

2. **Random Class Diagram Conform Data:** In Order to better evaluate functions and methods that are based on the base class diagram, we need a set of objects that conform to it. We can combine the RNG with generated builder classes from the class diagram to not only produce random data but also to create valid random objects. Note that these objects are very likely to not match any use case, as names are still random character sequences and values are only bound by the constraints imposed by the builders. Similar to *Random data*, although this data can be shown by the GUI-models this test data is unusable to optimize the user interfaces, as fields like user names or addresses would still only be random sequences of characters and would not represent typical usage.

3. **Random Constrained Data:** In order to create more applicable test data, we can introduce constraints that are tagged to specific attributes in the base class diagram

(*cf.* Section 3.5). The generator is extended by a function library that enables it to generate extended builders that not only can build objects, but also create random dummy objects. Thus we can produce objects that are bound to specific properties such as number ranges, string length, or amounts of linked objects.

4. **Random Use Case Specific Data:** User experience testing in particular relies on realistic data to provide plausible interfaces for the everyday use of a system. Therefore we can extend the generation of random constrained data with data sets for specific values such as names or parameters. Depending on the tagging, each parameter is either created at random or picked from a data set, thus assembling a new random object. Values themselves can also be assembled by using multiple data sets as input, enabling the creation of strings such as a concatenation of a title, a name, and a surname.

5. **AI-Based Data Generation:** LLMs are capable of producing datasets for specific data structures in various formats. The limits discussed in Section 4.1 apply here as well: Only a limited amount of data can be produced due to the limited context size of the used language model. Thus LLMs are well suited to find smaller edge cases rather than producing large data sets. Generating and training LLMs is very resource-intensive, a data set for test data should be created once and then used iteratively rather than creating a data set with each run of the generator.

So, using a class diagram, we can create a prototype together with random data with which the system can be tested and evaluated. However, as soon as we want to use data that is more precisely tailored to a specific use case, we need further models in the form of tagging.
Listing 6.13 depicts an excerpt of the tagschema used to define how test data should be generated for the tagged element of the class diagram. In this case, three strings can be tagged to an attribute: A default string (Line 7) that is used to set the attribute each time a test object is created. A prefix (Line 8) and a suffix (Line 9) that can be chosen from a list of strings stored in a file.

```
package de.monticore.montigem.tagging.tagschema;

import de.monticore.umlcd4a.symboltable.*;

tagschema GeneratorSchema {
    // ...
    tagtype StringEquals:String for Attr;
    tagtype PrefixFromFile:String for Attr;
    tagtype SuffixFromFile:String for Attr;
    // ...
}
```
TagSchema

Listing 6.13: Excerpt from the tagschema used to specify how data should be generated for a given class diagram.

An example is shown in Listing 6.14: The 'name' attribute of the 'Person' class is set to the Space character (" "). As a prefix, a name from a list is chosen at random, and as a suffix, a surname from a list is chosen. Resulting in a random list of reasonable names - rather than random sequences of characters.

```
package de.monticore.montigem.be.domain;

conforms to de.monticore.montigem.tagging.tagschema.GeneratorSchema;

tags GeneratorTags for Domain {
  tag Person.name with PrefixFromFile = "firstNames.txt";
  tag Person.name with StringEquals = " ";
  tag Person.name with SuffixFromFile = "lastNames.txt";
}
```
Tag

Listing 6.14: Tags for the 'name' attribute of the 'Person' class, setting up a random creation of a name-surname combination based on two name lists.

6.6 Modifying Model Generation: Variability

Up to now, we have demonstrated the capability of using a transformer to derive a set of models from a single underlying model. Nevertheless, during the development process, there might be a necessity to adjust the generated models or alter the general attributes of the produced models. In the following, we will take a closer look at the variability and adaptability of CD2GUI.

Variability is the ability of a software system to perform planned or anticipated changes to meet new requirements that arise after its design [GWT+13, GWT+14]. The variability has to be predefined in two ways: The locations where variation can occur (variation

points) [Jac97] as well as the process of how variability is performed (variability mechanism). Examples of variability mechanisms are inheritance, parameterization, template replacement, or generation [Jac97, Cle05]. The variability can be performed at design time, runtime or configuration time [GWT+14] (e.g. selecting between different versions of a component during the design or dynamically at runtime).
In the context of DSLs, one option to realize variability, is to define it in a *variation model*, that describes how the variation is handled on a base model. The variation model can be defined in a different DSL or the same as the base model. Both models are then processed and a product model is produced. The product model is the base model with the variability applied and is defined in the same DSL as the base model. This process is called *Variability Processing*. The whole process is shown in Figure 6.21. The product model can then be used instead of the software system's base model, allowing for variability at the model level [HMPO+08]. Further options to apply variability to the generation process are *template replacement* and the usage of *stereotypes*.

6.6.1 Template Replacement

CD2GUI provides the same set of models for each class and does not handle visualization of non-primitive attributes and associations type specific. CD2GUI uses a set of nested templates to generate GUI-models. Each template can be replaced by handwritten ones (variation points) using the manager-class `TemplateManager` in order to add custom handling of additional types (*cf.* Figure 6.20). the template manager provides a `Set` storing all templates that are used to assemble GUI-models. The manager provides the methods to add, replace, and get templates which are called from within the templates (*cf.* Listing 6.15). The template replacement is designed to be conditional: Based on the provided class, attribute, or association, a different template can be returned. We use this to pick fitting templates that match the use case defined by the domain model. Thus a template that iterates over class attributes can use fitting templates to generate GUI-Components that correspond to the attribute type. Hence a temperature parameter is visualized with a temperature gauge and a percentage with a progress bar.
The method `getTemplate(String oldTemplateDesignation)` returns a specific freemarker template. Additional conditions can be provided by providing further arguments:

- Class specific template `getTemplate(String templateDesignation, ASTCDClass clazz)`: If provided with a class, any template for that designation and that class will be used instead of the default template.

- Attribute specific template `getTemplate(String templateDesignation, ASTCDAttribute attribute)` If provided with an attribute, any template for that designation and that attribute will be used instead of the default template.

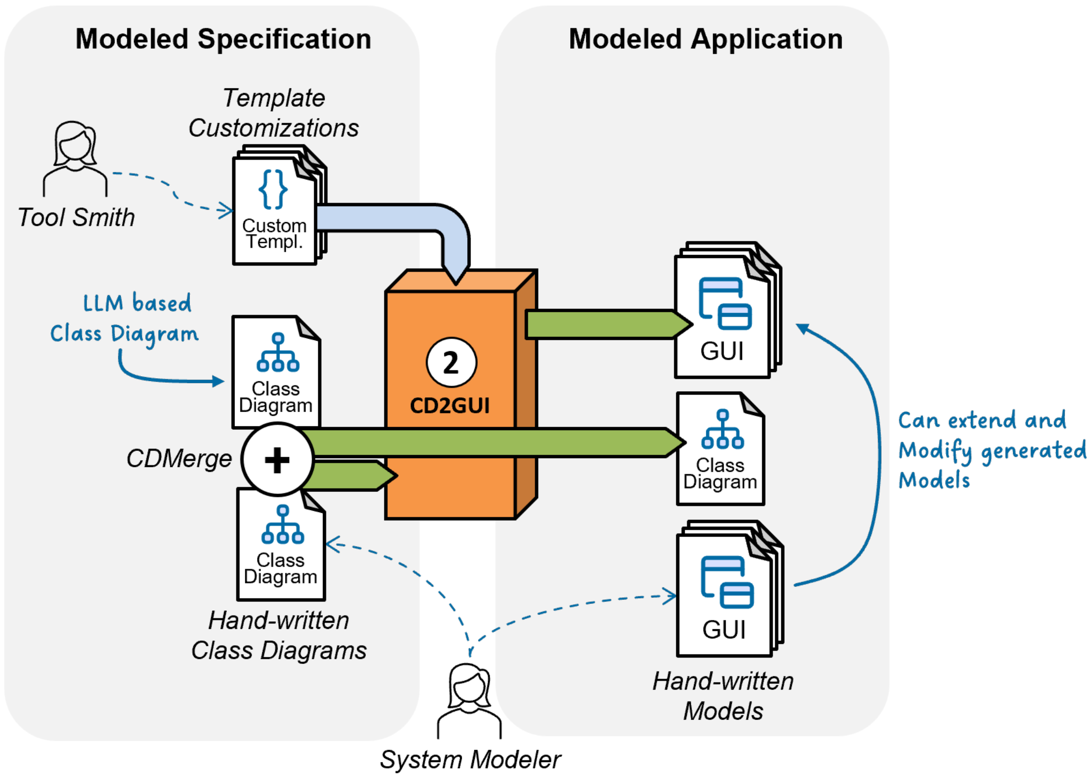


Figure 6.19: CD2GUI transforming CD4A models to GUI-models. CD2GUI can be configured to use custom templates (top left) in order to systematically modify the transformation.

- Association specific template `getTemplate(String templateDesignation, ASTCDRole role)`: If provided with an association, any template for that designation and that association will be used instead of the default template.
- Template Replacement for multiple elements `getTemplate(String oldTemplate, ASTCDClass clazz, ASTCDAttribute attribute)`: In order to replace a template if it is only used within a specific class, we can provide multiple elements such as class and attribute. In case an attribute name occurs in multiple classes, we can use this replacement to specify what attribute is replaced in which class.

Listing 6.15 shows the usage of template replacement in the template for the overview page. Depending on the class in focus, a corresponding template is used. In case no specific template is defined for that class the default template is used. In this case the developer chooses if he wishes to use the default table representation (*cf.* Section 6.2.2), or if he wants to use his own template *e.g.,* defining a chart or a dashboard.

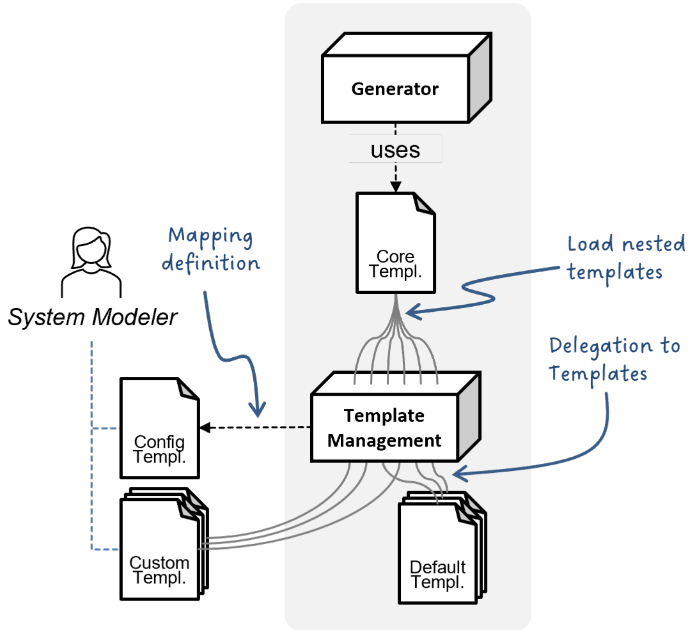


Figure 6.20: A system modeler can use a configuration template to change the mapping of templates from default templates to custom ones.

The developer can use template replacement within his own template, allowing for template reuse while keeping the variability of the template replacement.
As many of the MontiCore-based generators, CD2GUI can be provided with a configuration template (*cf.* Listing 6.15). The configuration template can be used to replace the templates used to create each GUI-model.
Similar to getTemplate described above we can define templateReplacement based on the class diagram elements that are provided. The method `replaceTemplate` defines rules for with class, association, or attribute a template is used that deviates from the default configuration. Similar to the definition above, rules for combinations of attributes and classes, or associations and classes can be defined. A template-specific rule can only be defined once for a specific template and a specific class diagram element. Two templates can not be set to replace the same template for the same class. However, a specific template can be replaced for different classes or different combinations of classes and attributes. In case of a conflict in configuration, the more specific configuration overrules the less specific one. E.g. getting a template for an attribute within a class will always return the template configured for an attribute within a class not the one for only the class, although both might be configured. Listing 6.16 shows a template defining the replacement of the `table-overview` with the `myTemplate` template. The replaced template is called in a template for the creation of the overview page. This mechanism

```
${name?uncap_first}@GemCard(
    title = "${name} Overview",
    component = ${name?uncap_first}GemRow(hAlign = "space-between",
        components = [
        <#include TemplateManager.getTemplate("table-overview",
            domainClass)>
    ])
);
```

Listing 6.15: Calling a template from within another template using the *template manager.*

can be used to introduce type-sensitive logic into CD2GUI. As Freemarker supports scripting and can run Java code, it can be used to switch between GUI components depending on the type or the attribute name, e.g. replacing a text field with a chart in case a specific type is provided to the template.

```
// Excerpt from configuration template
<#assign TemplateManager = tc.instantiate("cd2gui.util.TemplateManager")>
${TemplateManager.replaceTemplate("table-overview", new File("myTemplate"
    ).getAbsolutePath())}
```

Listing 6.16: Configuration Template provided to CD2GUI, replacing a template.

Template replacement is a mechanism that is useful in order to add a stronger domain-dependent focus on the generation process. The generic CD2GUI transfromer can be configured to produce user interfaces that are optimized to one specific domain: e.g. targeting financial dashboards and visualizations.
As this mechanism provides a systematic replacement of the templates used, it is not intended to modify individual models. In the following, we will take a look at further mechanisms that can be used to modify individual models.

6.6.2 Using Stereotypes to Customize Generation of GUI-Models

A primary advantage of CD2GUI is that it alleviates the need for expertise in GUI modeling, emphasizing instead on the construction of class diagrams. However, when system modelers employ template replacement, they reintroduce the demand for GUI modeling skills and, additionally, must acquire expertise in freemarker template definitions.
To mitigate this challenge, the generator can be tailored using stereotypes. This adaptation allows it to implement fundamental modifications GUI-model based on the given class diagram. As a result, the system modeler, by embedding information in root class

diagram through stereotypes, can change the result of the corresponding GUI-models. Similar to the tagging-based RBAC (*cf.* Section 6.5.1) stereotypes ensure that CD2GUI GUI models are generated in a different way. Depending on the stereotype, GUI components are ordered differently or left out.

Note that both the usage of stereotypes and template replacement can be used in conjunction. Stereotype usage targets the early stages of development, such as conceptualization and prototyping, whereas template replacement enables the developer to refine GUI-models in later stages of development. An overview of the stereotypes available in CD2GUI is shown in Table 6.4.

Key Attributes

Class diagrams are not limited in the number of attributes and associations of each class, but there is a limit to screen space and a hierarchy of relevance of class diagram elements to the user. Users tend to identify objects by a small subset of attributes: A user is typically identified by his username and not by his registration date. A booking might have a unique booking number that is more relevant for its identification than its booking status. When listing objects, identifying attributes must be prioritized in order to help the user find specific objects. At the same time, additional attributes that are irrelevant to the finding of an object should be removed from the overview pages. Thus, we introduce the concept of *key attributes*:

The system designer may use the stereotype `«key_attribute»` to denote an attribute as 'important'. Any lists containing objects of this kind will include this attribute. CD2GUI will generate a maximum of 8 columns from the class's attributes and exclude any extras since the interface lacks the space to display an indefinite number of data columns. Columns are selected based on the sequence in which they appear in the single underlying model. Any attribute designated as `«key_attribute»` will be displayed. This method can also accommodate over 8 columns if more than 8 attributes are marked. However, the resultant UI may exceed the screen boundaries.

The key attribute mechanism is applied to both the object list on the overview page and the linked objects list on the details page. The listed attributes of an object in the details page are not affected by this stereotype.

Hiding Classes and Attributes

There is a possibility that not all elements defined in the root class diagram should be visualized or editable in the UI. Common examples are sensitive data, such as the user password, or private data such as salaries or phone numbers. This problem can be addressed in part through role-based access control (*cf.* Section 6.5.1). Restricting access to classes or attributes still leaves 'empty' user interfaces, as the UI will leave blanks to data that is restricted (empty tables, fewer rows, fields left blank).

In order to hide an attribute from all users irrespective of their user permissions, a mechanism is provided to either exclude it from the generated GUI-model or stop its generation entirely. The stereotype `«invisible»` can be assigned to both classes and attributes. For classes, this stops CD2GUI from producing type-specific pages for that class. For attributes, it prevents them from being displayed on any type-specific page.
When the system designer intends to conceal an attribute solely on the overview page (see Section 6.2.2), the stereotype `«hide_in_overview»` can be used. This will remove the attribute from the overview page of the relevant class.
An attribute designated as `«key_attribute»` and marked as `«invisible»` or `«hide_in_overview»` will not be included in the corresponding GUI-models and consequently will be hidden from the UI.

6.7 Modifying Generated Models: Adaptability

The previous section presented a mechanism on how to systematically modify the process that generates GUI-models. In this section, we will take a closer look at the mechanism that enables the modification of models once they are generated. This Section focuses on mechanisms to modify generated models, the inclusion of the TOP-Mechanism is discussed in Section 7.7.

6.7.1 Adaptability

Within this work, we refer to adaptability as the ability of a software system to be adjustable in its behavior or structure in response to changes in its use case or in its requirements. This often means that the software can evolve or change post-deployment without requiring significant redevelopment.
We use similar mechanisms as presented in Section 6.6. The system modeler is provided with a set of GUI-models and is required to adapt a generated 'generic' model to better fit the use case of the target domain of the application (e.g. change the visualization of data to a more comprehensive model). Each model is defined to be assembled of a set of exchangeable GUI-Components (*cf.* variation points Section 6.6). In the following, we will present the mechanism governing the adaptation of models.

6.7.2 Handwritten GUI-model Extensions

This section deals with the adaptability of GUI models of GUIDSL v2 syntax. GUIDSL v1 models do not support references to other GUI models and therefore do not support this mechanism.

Operations

In order to enable the complete modification of an arbitrary GUI-model to any other (valid) GUI-model, we define a set of operations to describe GUI modifications. The fundamental concept of delta modeling languages has already been explored by Haber et al. in [HHK$^+$13] and [HRRS12]. Instead of defining an additional language to define deltas upon a GUIDSL model, we incorporated many of the presented concepts into GUIDSL v2 directly.

1. **`replace`** - replaces an existing component with another one.
2. **`remove`** - removes a component from the model without replacement.
3. **`before`** - adds a new component before an existing one.
4. **`after`** - adds a new component after an existing one.

These operations enable modification at any variation point in the model. We can define each GUI-component, guard-symbol, and iterate-symbol of a GUI-model with a unique id (cf. Section 5.2), every unique identifiable element in a GUI-model can be targeted by the operation above and thus servers as a variation point.
`replace` is an operation that removes a portion of the user interface and inserts a new component in its place. Besides replacing it also implements removal - by replacing with an empty statement, and adding constructs - by replacing an existing construct with itself and a new component. However, using replace to perform removal or addition is unintuitive, thus we introduce `remove`, `before`, and `after`.
The variation points may represent small parts of the user interface, such as a single button or text element, but could also be a collection of components up to an entire web page. This makes the operations a powerful tool for the modification of the GUI, as it can be extended or reduced at any point.

In order to adapt a generated model, we have a second hand-written model that defines which components of the generated model are modified and how they are modified. In the following, we refer to the generated model as the *Base Model* and to the model containing the adaptation operation as *Adaptation Model* (*cf.* Figure 6.21).

Identifying the Base Model

We define a GUI model as the target of the adaptation model. Thus, we need a mechanism to identify the base model in question. Our solution is to import and reference the name of the target in the adaptation model. The fully qualified model name is unique, which guarantees proper target identification.

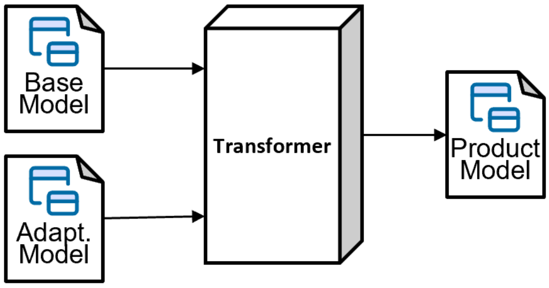


Figure 6.21: Adaptability for GUI-models

Implementing Delta definitions in GUIDSL v2

The hand-written model defining the adaptation of the base model is a gui model as well. Therefore, we have to extend the GUIDSL grammar, in order to enable the system modeler to define the operations within the GUIDSL syntax. The implementation of adaptability in the GUIDSL grammar is shown in Listing 6.17. The `GUIComponentAdaptation` realizes the identification of the target model. The syntax is similar to Java `extends` and has a similar purpose. Operations `replace`, `before`, and `after` are all summarized in one construct, since they are very similar and can be represented by a single grammar rule. These operations are applied to a variation point referenced by its `id` (`Name@Variable`) and add or replace a new GUI part (`GUIArgValue`), i.e. a `GUIComponent`, `GUIGuard`, or `GUIIterate` construct. `Remove` on the other hand does place a new GUI part and thus has a separate syntax and grammar rule specifying which portion of the user interface has to be removed. The check whether the operation is properly defined and targeted variation points exist takes place at generation time and is performed by the GUIDSL. The new constructs reuse the grammar rules of GUIDSL, are intuitive and easy to learn for a modeler familiar with GUIDSL. An example of an adaptation model is shown in Listing 6.19. It modifies a base model shown in Listing 6.18, using extended grammar. The resulting product model is shown in Listing 6.20. The base model (Listing 6.18) defines the simple page `BaseModel` (Line 2) that contains 3 constructs: `comp1, comp2, comp3` (Line 3..5). The page consists of two texts and a simple button. The adaptation model removes the construct with the ID `comp1` (Line 5), replaces `comp2` with a text (Line 6), and places another text after `comp3` (Line 7). Therefore, the resulting product model (Listing 6.20) is still named the original base model (`BaseModel`), but now contains the new constructs `new2` and `new3` before and after the unchanged button `comp3`.

6.7.3 Adaptability Transformation

The adaptation transformation can be performed by traversing the AST. The nodes are strictly hierarchically ordered without any interdependence, allowing for changes on a

```
symbol GUIComponentDeclaration
    implements Function =
  (["component"] | ["page"]) Name "("
    (GUIParam || ",")*
  ")"
  GUIComponentAdaptation?
  GUIBlock?;

  GUIComponentAdaptation =
    "adapts" target:
      Name@GUIComponentDeclaration;

  // Operations
  GUIAdapt = position:
    ["before" | "after" | "replace"]
    Name@Variable ":" GUIArgValue;

  GUIRemove = "remove" Name@Variable;
```

Listing 6.17: The GUIDSL grammar additions

```
// ...
page BaseModel() {
  comp1@Text(value="Example UI");
  comp2@Text(value="Example Text");
  comp3@Button(text="Click me");
}
```

Listing 6.18: An example *base model* in GUI-DSLv2.

node without affecting other nodes than its children. Consequently, adding an AST node or swapping the node at the variation point can be used to effectively perform the adaptation transformation. The process can be seen in Figure 6.23, in this case, a table component is changed into a pie chart component.

ASTs can be efficiently processed via traversal. Since both the base model and the adaptation model can be represented as an AST, a combination of traversing both ASTs is sufficient to perform the adaptability transformation. The process is shown in Figure 6.22.

In order to create the product model, both ASTs are traversed. For each operation in an adaptation model AST, we store the corresponding target construct and continue by traversing the base model AST.

Once the target construct of the operation is found, the operation can be performed on the same AST that was traversed to find the target of the operation. After the

```
// ...
component VarModel() adapts BaseModel {
  remove comp1;
  replace comp2:new2@Text(value="repl 2");
  after comp3:new3@Text(value="after 3");
}
```

GUI-DSL v2

Listing 6.19: An example of a *adaptation model* for Listing 6.18

```
// ...
page VarModel() {
  new2@Text(value="repl 2");
  comp3@Button(text="Click me");
  new3@Text(value="after 3");
}
```

GUI-DSL v2

Listing 6.20: Resulting *product model* based on Listing 6.18 and Listing 6.19

operation is performed, the traversal of the adaptation AST is continued to perform more operations that might be defined in the adaptation model. Editing the base AST directly with each operation means that no conflicts between operations need to be resolved. The approach used applies the operation sequentially, which means that the conflicts either arise during the traversal, e.g., the target of an operation is non-existing, or are resolved as a result of the used strategy, e.g. adapting a target twice. In the first case, a warning is issued and in the second case, the behavior is normal since the target can be adapted several times. In either way, no additional detection or resolution of conflicts is needed.

The traverser for the implementation of the adaptability transformation is already by MontiCore, so only the methods for the handling of the nodes in question need to be implemented - for the adaptation AST `GUIAdapt` and `GUIRemove` and for the base AST all nodes that can contain a `GUIComponent`, `GUIGuard`, and `GUIIterate`, i.e. `GUIBlock`.

When traversing the adaptation AST, the operation and respective parameters can all be found directly in the `GUIRemove` and `GUIAdapt` AST nodes. Those can then be saved for a traversal of the base AST, where the operation is performed.

The last part of the grammar to consider is the imports. The adaptation model should be able to use different components as the base model, the product model should also still be valid in case the base model imports change. To realize this, all imports that are not already present in the base model are copied from the adaptation model. This means that when the base model changes, potentially missing imports are still added since they are present in the adaptation model. The addition of imports can be easily handled in the `ImportStatement` nodes.

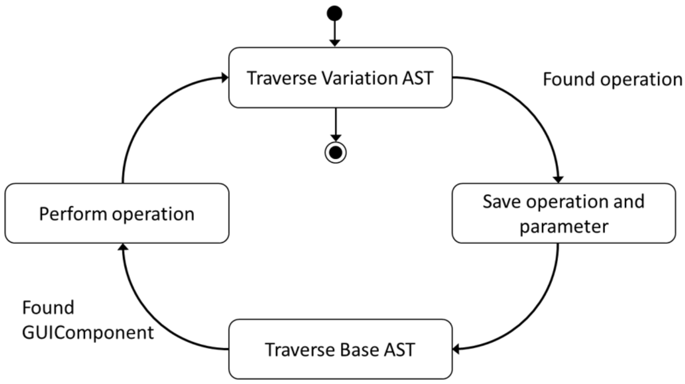


Figure 6.22: Adaptability Transformation

This general design makes this grammar extension very robust. New grammar constructs are unlikely to affect this approach, which only needs to be changed when a new construct needs to be supported. Changes to the existing grammar do not require an update unless they specifically target the constructs used in this extension.
Issues in the adaptation model can also be caught quite easily. The adaptation model itself can already be validated by the parser. Inconsistencies between the base and adaptation model, e.g., a reference to a non-existing `id`, are also found early with the help of the symbol table.

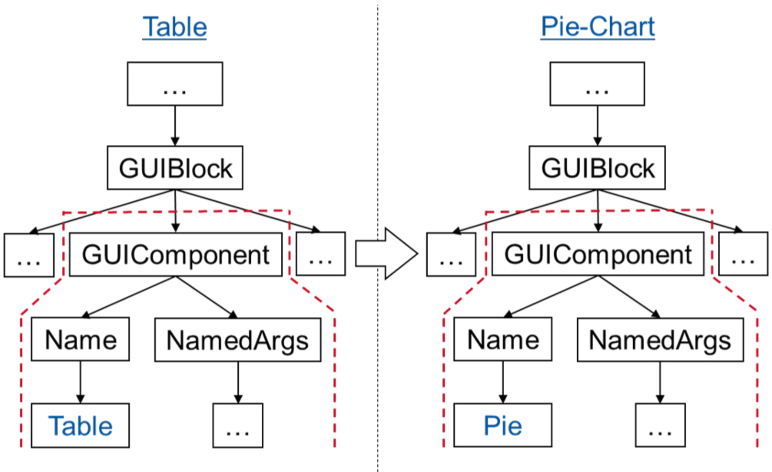


Figure 6.23: Using the tree structure to change sub-trees and alter the AST of the GUI-model

6.7.4 Edge Cases

In GUI-models the order in which components are defined is relevant for the produced user interface. This also holds for the operations to modify another model. In the following, we will take a look at a few cases in which the order of operations changes the outcome.

```
// ...                                   GUI-DSL v2
page BaseModel() {
  c1@Text();
  c2@Text();
}
```

Listing 6.21: Base model defining two GUI Components

```
// ...                                   GUI-DSL v2
  remove c1;
✗ replace c1: c3@Text() ERROR
// ...
```

Listing 6.22: Attempt to replace a component after it was removed.

The targeted elements must be present in the targeted model. Therefore, in case we try to call any operation on a GUI element that was already removed (*cf.* Listing 6.21, Listing 6.22, an error will be thrown.

```
// ...                                   GUI-DSL v2
component VarModel() adapts
    BaseModel {
  replace c1: c3@Text()
// ...
```

Listing 6.23: Replacing component c1 with c3

```
// ...                                   GUI-DSL v2
component NewVarModel() adapts
    VarModel {
  replace c1: c2@Text()
// ...
```

Listing 6.24: Replacing component c1 with c2

In case two models target the same model to adapt, the order in which they are interpreted is relevant. Depending on which of both models Listing 6.23 and Listing 6.24 is interpreted first, the resulting component will be 'c2' or 'c3.' An error will be thrown because there is an ambiguity here.
The adaptability transformation is set up to match a set of handwritten models to a set of generated ones. This excludes adaptations from handwritten models to other handwritten ones or adaptations that span over multiple models (e.g., adapting a model that already adapts another). Although these adaptations would be feasible, the method implemented here focuses on enabling the modification of the generated artifacts and, therefore, does not yet provide that functionality.

Type	Supported
MyClass 1 ↔ 1 MyOtherClass	✗
MyClass 1 ↔ 0..1 MyOtherClass	✗
MyClass 1 ↔ * MyOtherClass	✓
MyClass 1 → * MyOtherClass	✓
MyClass 1 → 0..1 MyOtherClass	✓
MyClass 1 → 1 MyOtherClass	✗
MyClass * ↔ * MyOtherClass	✓
Enum	✓
Interface	✓
Abstract Class	✓
Inheritance	✓
Basic data types	✓
Zoned Date Time	✓
Optional<Class>	✓
Packages	✗
Generics	(✗)[1]
Annotation	✗

Table 6.3: Supported class diagram elements by CD2GUI. 1: Only Lists and Optionals are supported, not Generics in general.

Stereotype	Function
`«invisible»`	Element will not be shown at all
`«hidden_in_details»`	Element will not be shown in the details page
`«hidden_in_overview»`	Element will not be shown in overview
`«key_attribute»`	In case of limited screenspace element will be prioritized

Table 6.4: List of stereotypes available in CD2GUI, adjusting what elements are shown in the UI.

Chapter 7

Generator Framework for Enterprise Management: MontiGem

In Chapter 2 we introduced a methodology that is based on three major transformation steps. The first is a transformation from natural language to a data structure model (LLM4CD Chapter 5) the second, is a transformer that produces GUI-models based on the provided data structure models (CD2GUI Chapter 6) and the third being a generator that uses both GUI-models and CDs to produce a web application:

Contents

MontiGem (**G**enerator for **E**nterprise **M**anagement)[AMN+20, MNN+22, DHM+22, DGM+21, GMNR21, GMN+20] is a generator framework for the engineering of web-based information systems (*cf.* Figure 7.2). It synthesizes the source code for a web application with a three-tier architecture, that uses a TypeScritpt-based client (Angular) in combination with a Java-based Server (Apache TomEE). The framework has proven itself in several use cases, five of which are described in [BGK+23b]. An in-depth discussion focusing on the perspective of this thesis is presented in Chapter 9. MontiGem (Figure 7.1) uses textual models as input, that are parsed to Abstract Syntax (AST). This abstract representation is transformed and extended, processing the information of each model into the target code. Finally, the ASTs are passed to template engines, that produce the target code, which is used in combination with a static RTE of the web application which is provided by the framework.
These models include three kinds of UML *class diagrams* (*cf.* Section 3.4) to define data structure of the application itself, to define views upon the data, and to define addditional commands between server and client. Also models to define the *graphical user interfaces* (*cf.* Section 5.1), and *OCL* models that define constraints for data validation within the client and server of the derived application. Additional information can be appended to the models via *tagging* (*cf.* Section 3.5) models.

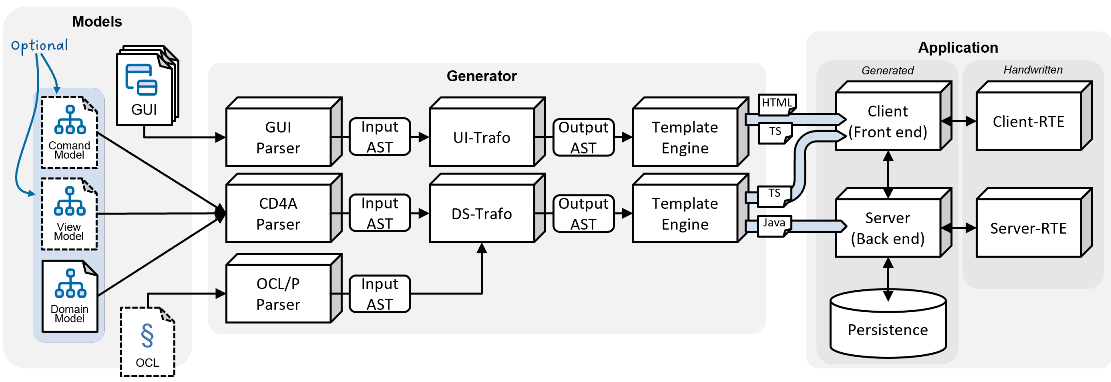


Figure 7.1: Architecture overview of MontiGem. Parsers that were generated with MontiCore are used to convert textual models into abstract syntax. Input AST is transformed for the abstract target implementation and finally transformed into the target source code of the desired GPL. The generated code is only a part of the target application, next to the RTE in both server and client.

The term MontiGem-framework describes a collection of components that are not limited to the generators. The generated code requires a run time environment (RTE) and model-independent source code, in order to be able to produce a running web application. The code generated by MontiGem is used as part of an application implementation (*cf.* Figure 7.2). The generators provided are used in the respective front-end and back-end build processes to provide the target source code defined by the input models. This generated code for both the front end (FE) and back end (BE) can be extended and overwritten by handwritten implementation. Additionally, there is static application source code, that is not based on any of the input models. This includes generic application configurations, but also common data structures e.g. user management that is required by any web application. In the following, we first take a look at an overview of the *Target Application* (*cf.* Section 7.2), next we present both generators: *Data Structure Generator*(*cf.* Section 7.3) and *Generator for Graphical User Interfaces* (*cf.* Section 7.4). The generated code works with two RTEs, the *Client Run Time Envoronment* (*cf.* Section 7.6) and the *Server Run Time environment* (*cf.* Section 7.5). Finally, we present a mechanism to extend generated code with handwritten code (*cf.* Section 7.7).

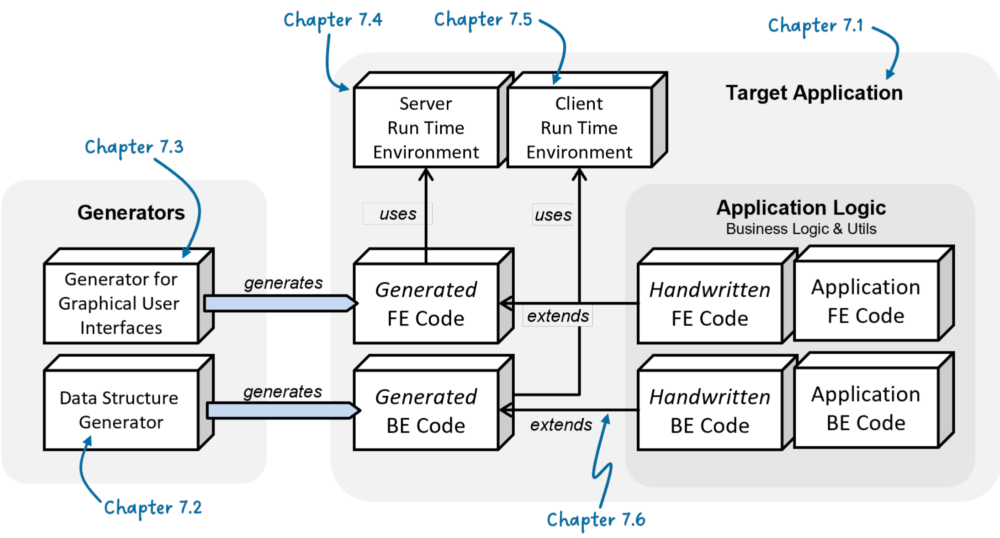


Figure 7.2: Key componentes of a MontiGem-generated application.

7.1 Research Method

The development of the MontiGem generator framework follows the case study approach. The generator is developed for several use cases [BGK+23a, NGM+24, DGM+21, MNN+22, GHK+20]. The generator is initially developed for the use case MaCoCo a data-centric information system for the controlling of finance, staff, and projects of the MaCoCo. The chairs of the RWTH Aachen University. We collect data on the generator performance and gradually improve its usability and generalizability, enabling the framework to be used more in general for further use cases.

7.2 Target Application

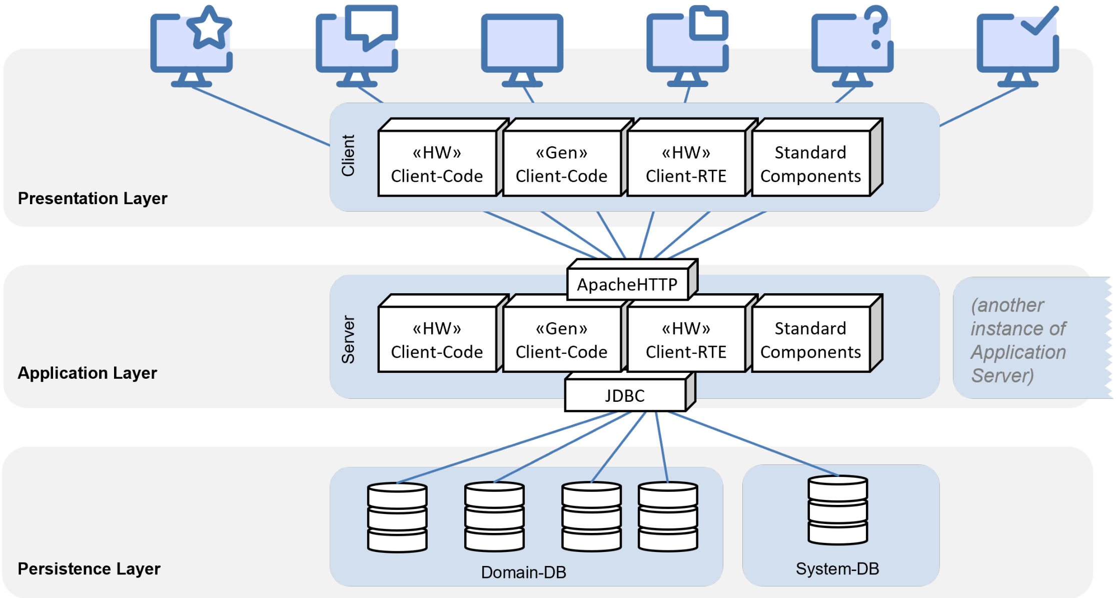


Figure 7.3: Three-tier architecture of MontiGem

We refer to MontiGem as the generator framework that produces the target source code and not the resulting web application. The application based on it is either referred to as *web application* or *target application*. The generator produces domain-specific components that work in conjunction with the static run-time environment (RTE). Thus, the architecture of the target application is predetermined by the generator configuration. Similarly to MontiDEx [Rot17], it is based on a three-tier pattern [Eck95] (Figure 7.3). In MontiGem the application layer resides on the remote server, whereas in MontiDEx is run as a Java Application on the client system.
The first layer is the *Presentation Layer*. In our client-server architecture, it represents the client part and the graphical user interface with which the user interacts. Multiple clients can connect to the application server via an Apache-HTTP server. The presentation layer contains user interfaces generated in HTML and Typescript. The generator framework produces a thin client, reducing the logic in the presentation layer to a minimum. Handwritten extensions adapt user interfaces and add small process flows to the front end, which would be harder to define in corresponding models for the user interfaces. The RTE in this layer consists of generic web application components and utility

classes, where standard components are the default building blocks of the user interface such as generic tables, layouts, and buttons.
The second layer is the *Application Layer*. It contains a TomEE server that runs within its own docker container and thus has the potential to be configured to run as multiple instances at the same time. Open-source systems such as Kubernetes (K8s) can be used to automate deployment and load balancing between deployed containers [NK20, Ren15], making the approach scalable, as it provides the ability to distribute incoming network traffic over a group of back-end services. The application core runs on the generated data structure defined through the input models. Handwritten extensions can be used to add more business logic to this layer. The RTE consists of several classes and utilities that are needed to run any server back end (e.g. authentication or API-Services). These components contain custom implementations for a MontiGem-based Server, but also third-party frameworks such as Apache-Shiro and JAX-RS.
The third layer is the *Persistence Layer*. MontiGem-generated applications are provided with a generated database and corresponding persistence management. The RTE of the persistence layer contains database connectors such as JDBC and a persistence API (Hibernate) that enables the application to access the database without conflict. The generated access to the database is typically extended with handwritten code to optimize performance or to add additional checks for validity or security. Next to the collection of domain-dependent databases, a system database is generated that stores system-wide information, such as global configurations for the system.
The core of the system consists of two generators that are responsible for the creation of the user interface (GUI Generator) and the infrastructure for data management (DS Generator). In combination, both generators create a custom web application based on the provided data and UI models.
Figure 7.4 provides an overview of the basic artifacts used in the target code that was generated with MontiGem. The generator framework produces a web application, as such we can divide the code into front end Ⓐ, back end Ⓑ and database Ⓒ. The generator for the user interfaces Ⓓ and the generator for the data management infrastructure Ⓔ are not part of the finished web application and can be managed in separate projects. The code in the front end as well as in the back end can be adapted and personalized for the application by handwritten code. In the front end, this is typically a customization of the user interface or its underlying logic Ⓕ, and in the back end, the DTOLoader Ⓖ, a class defining how data is provided to the client is often extended with business logic as well.
We can divide the implementation itself into three parts.

1. First, the domain-dependent code (below the dashed line). These artifacts are related to the current use case and might change from application to application. It contains business logic, such as case-specific data structure and user interfaces, as depicted in Figure 7.4.

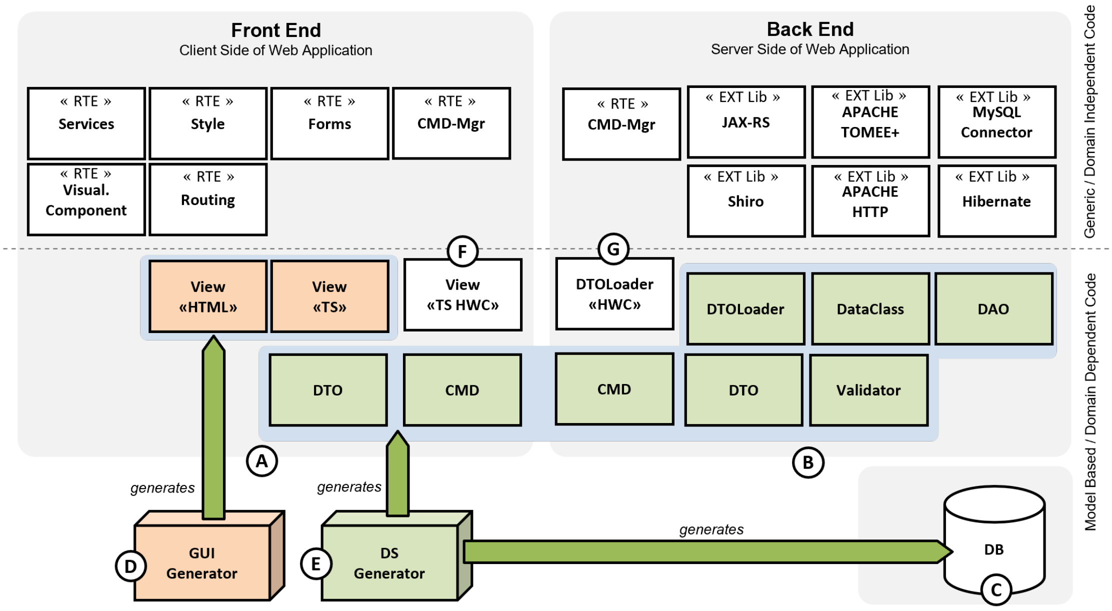


Figure 7.4: MontiGem generator output nested in RTE and External Libraries. DS Generator (E) generates data management infrastructure, GUI generator (D) creates user interfaces, both can be extended with handwritten code such as (F),(G). The target application is segmented into front end (A), back end (B), and database (C).

2. Above the dashed line, generic (domain-independent) code is shown. We split the generic code into further groups: It contains the second part: RTE elements such as basic visual components, APIs, and elements that are typically reused between web app implementations. This code is not produced by a generator and is provided as-is by the framework. It is marked with `«RTE»` in Figure 7.4.

3. As a third part, we identify external libraries that are used by the web application. This group contains, among other libraries, the Tomee[1] Server Framework, Java Persistence API[2] and Security Framework[3]. This group is marked `«EXT Lib»` in Figure 7.4.

[1]tomee.apache.org
[2]hibernate.org
[3]shiro.apache.org

7.3 Data Structure Generator

In the following, we will focus on the domain-dependent aspects of the generator (*cf.* Figure 7.4 below the dashed line). We will take a closer look at the classes that are generated by the data structure generator E. The GUI-generator D will be presented in the next section (Section 7.4) followed by a section regarding the RTE (*cf.* Section 7.5). MontiGem produces a large collection of artifacts that govern the operation of both the server and the client of the web application. In order to maintain readability, we do not present every class that is generated by MontiGem, but focus on the most important ones. The key classes the data-structure generator produces (as shown inFigure 7.4) will be explained in detail in the following sections:

- **Data Class:** Basic Transformation from the Domain model into a Java class with a simple extension for getters and setters.
- **Data Access Object** (DAO): Class that provides database access for a specific type.
- **Data Transfer Object** (DTO): Class that encapsulates an object for data transfer so that all or at least only a few associated objects are loaded as well.
- **Command** (CMD): Class that provides means to trigger events in the back end from the front end. For each class in the domain model a set of commands is generated to load data and to provide basic CRUD operations.
- **DTOLoader**: Class that creates an DTO-Instance by using a corresponding DAO to load data from the Database. The DTOLoader is typically used in the context of a command.

7.3.1 Domain Model

MontiGem provides multiple classes for every class defined in the input class diagram (domain model). Within our tool chain MontiGem is provided with the root class diagram, in order to produce a corresponding data structure. In this section, we will take a closer look at the classes generated by the DS-Generator (Figure 7.4 E). We will use the simple person class (*cf.* Figure 7.5) as an example input throughout the upcoming sections:
Based on the domain model, MontiGem generates infrastructure for the server and client. Figure 7.6 shows the different components that are generated in order to access and modify data in the client database. On the client DTOs are generated and used to store data within the generated commands. The commands are generated for both the server and the client and are used to run dedicated logic on the server. The data contained in the commands is loaded into matching DTOs at the server. DTOLoaders are used to map the data that is accessed via DAOs to these DTOs.

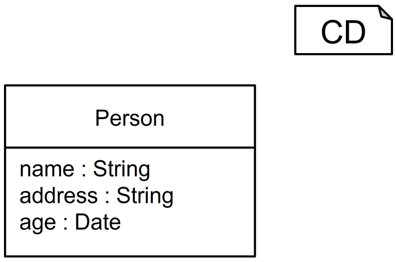


Figure 7.5: Class diagram defining a simple class for a person.

Dataclass

MontiGem produces a data class as shown in Figure 7.7, for each class defined in the domain model. Each generated data class contains additional attributes and methods and is structured as follows:

- **Database Annotations**: Generated annotation for the Persistence API. Based on these annotations Hibernate (or a corresponding framework) creates tables in a relational database. The annotation `@Entity` causes the creation of the corresponding table for Person and `@Audited` creates an observer for that table. Data classes are used to define the database schema.

- **Attributes**: Any non-derived (*cf.* Section 3.4.3 attribute defined in the domain model will be added as a private attribute to the data class. Additionally a protected attribute for a unique database `id` is added that serves as a primary key in the database table.

- **Constructors**: MontiGem adds default constructors. Note that MontiGem also generates builder classes (`PersonBuilder`) that use validators, and should be preferred to these constructors if possible.

- **Basic Object Methods**: `merge()` and `mergeWithoutAssociations()` can be used to update existing objects with a new set of values without creating a new database entry, by merging both objects. The existing values are overwritten by the new ones. The `equals()` method provides a simple way to compare two objects of this data class type.

- **ToString methods**: In the context of the generated web information system, types are often displayed directly on the user interface, therefore methods that provide 'human readable' names are added to the data class. These names can be defined and overwritten by stereotypes in the domain model. We differentiate between these methods and the normal `toString()` as the names used for the data class often differentiate from those displayed on the user interface.

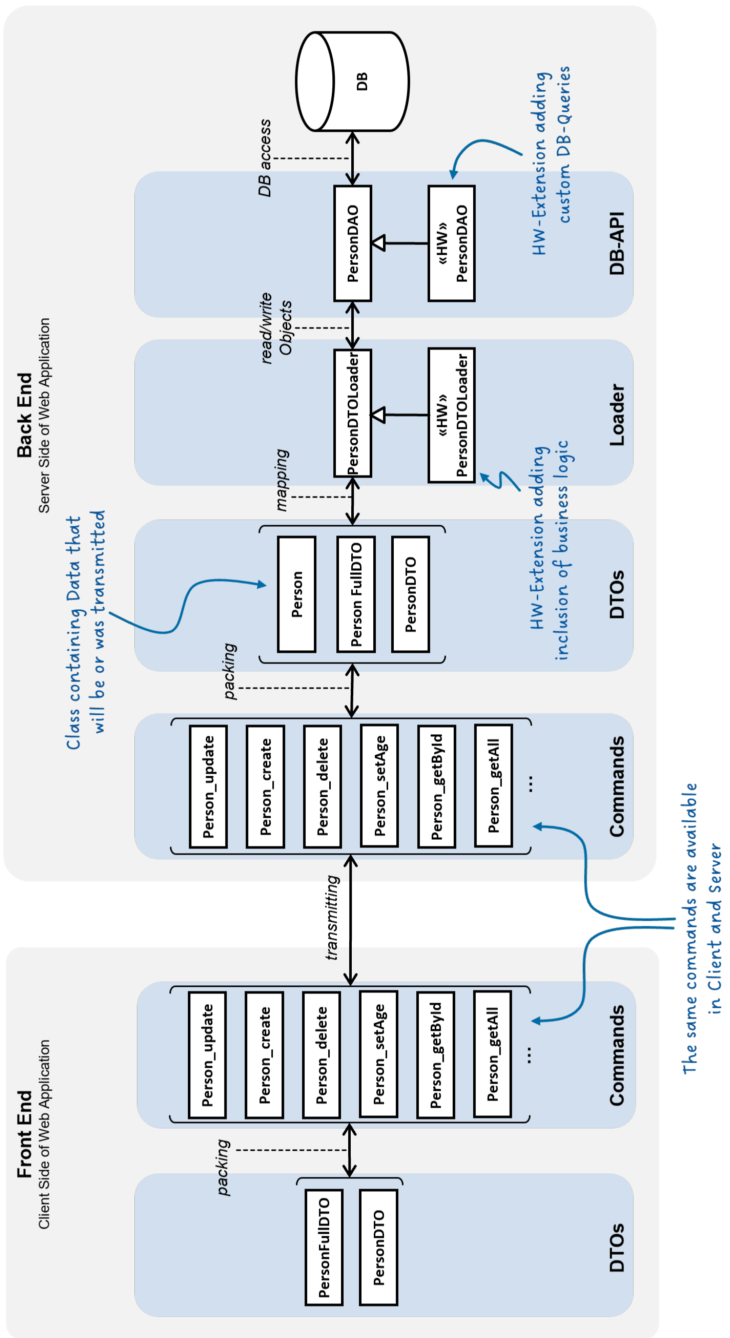


Figure 7.6: Relationships between generated components. A data can be accessed from the front end through multiple classes: An object from the database is accessed via a DAO, and a DTOLoader transforms the object into a lightweight DTO. The DTO is wrapped within a command, is serialized and sent to the client. The client receives the command and deserializes a DTO, that can be processed and visualized by the UI.

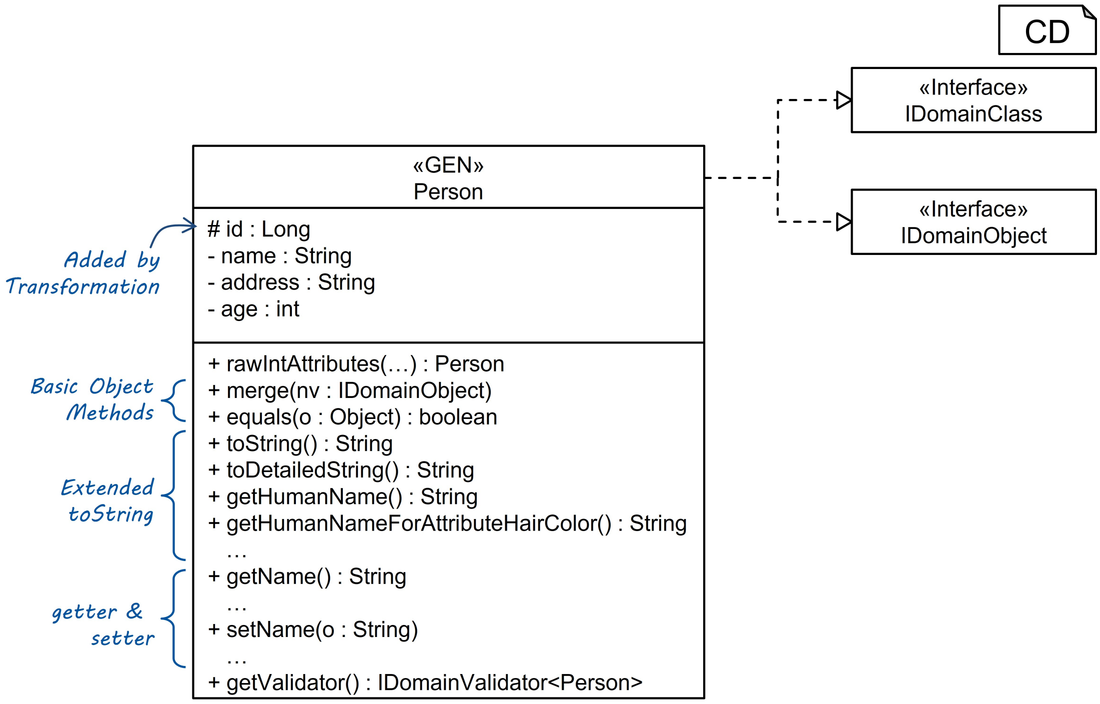


Figure 7.7: Generated data class for Figure 7.5. Data class implements two interfaces `IDomainClass` and `IDomainObject` handling generic operations like merging.

- **Getter and Setters**: These are default methods needed to set and get values of the attributes of the data class. The attributes can not be modified directly as they are declared with the `private` access modifier.

- **Validator** Provides the corresponding validator for this data class. The validator is a separate class that is generated by MontiGem and checks the validity of an object of this data type.

Data Access Objects

The **D**ata **A**ccess **O**bject is the interface between the data class and the database. Mapping objects to a database is a challenging task. The Java Persistence API (JPA[4]) is a specification that allows the developer to work directly with objects rather than SQL statements. There are several implementations for JPA, such as EclipseLink[5] or

[4] `https://www.oracle.com/java/technologies/persistence-jsp.html`
[5] `https://eclipse.dev/eclipselink/`

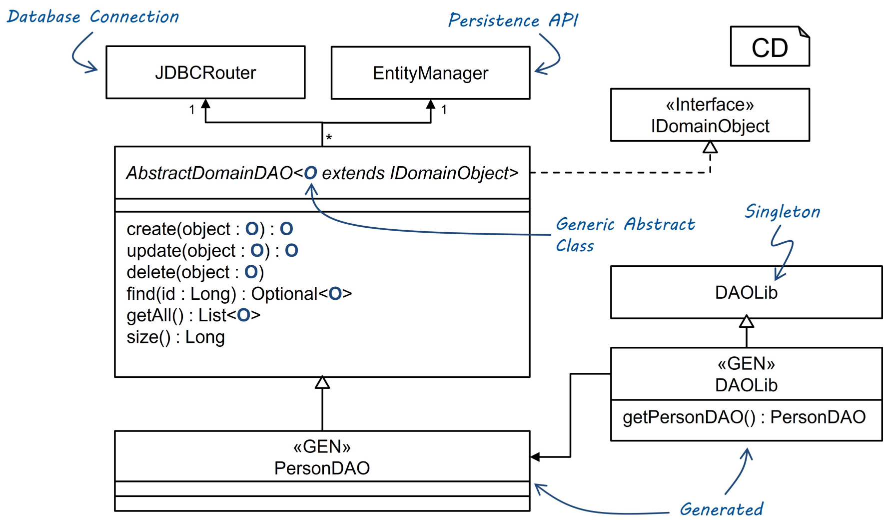


Figure 7.8: Generated data access object `PersonDAO` for Figure 7.5. The DAO extends the generic class `AbstractDomainDAO`, which contains multiple methods to access the database.

ApacheOpenJPA[6], in MontiGem Hibernate[7] is used.
For each class `X` in the Domain model a DAO-class `XDAO.java` is generated. Every DAO extends the abstract generic class `AbstractDomainDAO`. The DAO uses a Java Database Connectivity (JDBC[8]) router that can be configured via a specific JDBC driver to access a database. Connections to several database types such as Access, FileMaker, SQLite and many more are supported. There are implementations of MontiGem using both MySQL and PostgreSQL connections [ANV+18]. Next to the database connection, the JPA (Hibernate) has to be used in order to enable persistent database operations.
The DAO provides the developer with several methods to perform database operations, each method is overloaded multiple times to match the multiple use cases the class is used in. The example Figure 7.8 only shows a selection of representative methods.

- **Creating new objects:** The DAO provides several methods to store new objects in the database. There are options to create objects within the currently running persistence context or as an isolated object. Upon successful creation of the object the DAO will return an updated object containing the Database ID.

- **Updating an Object:** The provided update methods will check if an object exists. If so it will use the merge method defined in the data class to update the object.

[6] `https://openjpa.apache.org/`
[7] `https://hibernate.org/`
[8] `https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/`

If it does not exist, the corresponding create method is used.

- **Retrieving an object:** The DAO provides multiple methods to retrieve objects from the Database. To optimize performance, objects can be retrieved without any associated objects, as well as a specified depth of the association tree. This is especially useful if only parameters from the directly targeted object are needed and not of associated ones. As the retrieval of data from the database is on of the most commonly used operation, there are many specialized and overloaded methods for these tasks:
  - Finding an element for a specific id
  - Finding any element of a specific type
  - Getting all elements of one specific type
  - Getting all elements of one type for a List of IDs
  - Methods for Lazy Loading
  - Methods for Load eager
- **Deleting an object:** The DAO provides two options to delete an object. Deleting the object only if no other objects are affected. Delete the object and any related ones, keeping the database consistent. Note that in the default configuration, both methods run the same code, as there is no specification on which elements must not be deleted and which ones are fine to remove in a cascade.

Builder

MontiGem creates for each class in the domain model a builder class. The generated builder is based on the builder design pattern [Zur18, HH13], and decouples the object creation algorithm from the system. It allows to add new functionality to the creation process. MontiGem adds a validation step for each created object. The builder itself (Figure 7.9) is generated based on a corresponding class in the domain model. Thus it contains an attribute and a setter and getter (omitted in Figure 7.9) for each attribute of the original class. The generated setters support method chaining, thus keeping the implementation for the creation of objects with many attributes readable Listing 7.1. The `build`-Method produces an instance of the targeted class after using the generated validator to check all provided data.

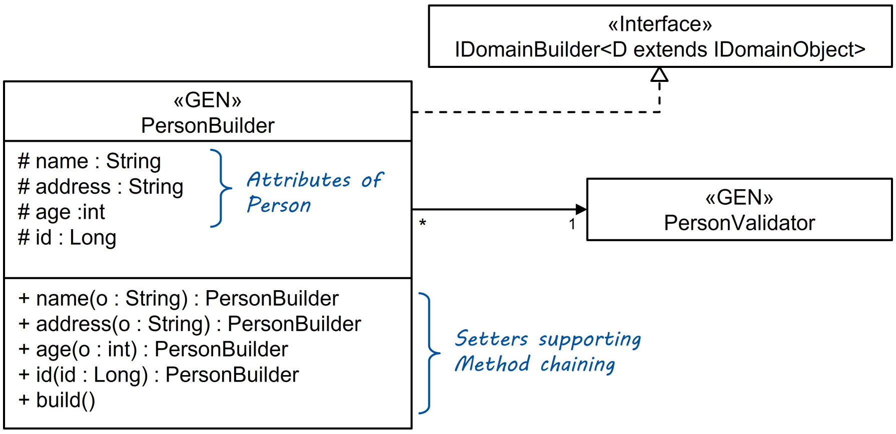


Figure 7.9: Generated Builder `PersonBuilder` for Person-Class (*cf.* Figure 7.5). The builder can hold the same attributes as the generated data class and provides chainable setters to set those. Before returning an instance of the data class, it uses the generated validator to check the input.

```java
PersonBuilder buidler = new PersonBuilder();
Person person = buidler
    .name("Sherlock")
    .address("Baker street 221b")
    .age(60)
    .build();
```

Listing 7.1: Usage of PersonBuilder to create new Person object

Validator

MontiGem creates for each class a validator (Figure 7.10). Each validator provides methods to validate each attribute, association, and additional constraints that can be defined by an OCL model.

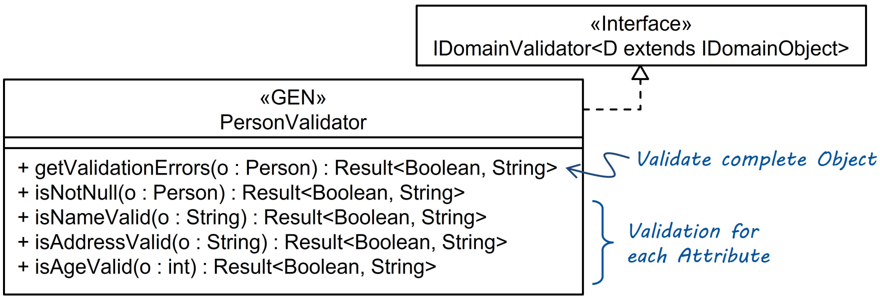


Figure 7.10: Generated Builder PersonValidator for Person-Class Figure 7.5.

```
context Person inv name:
    name.length() <= 50;
    shortError: "Max. 50 characters";
    error: "The name (" + name + ") is longer than 50 characters";
```
OCL/P

Listing 7.2: Example of an OCL constraint limiting the length of the name attribute of a person to 50 characters.

Additionally, an overloaded method to validate the complete object (getValidationErrors()) is generated and used on the builder class. The validator returns its result as an instance of the generic class Result<OkType, ErrTyp> that in case of the generated validator is implemented as a boolean indicating the success or failure of the validation and a string containing corresponding which contains error messages if necessary. The default generated validation checks for the type of the object and if an attribute or a linked element is a mandatory object, whether it is null. Similarly, it checks the cardinality for linked objects.

```
public Result<Boolean,String> isNameValid(String value){
  if (value == null) {
    String msg = "Parameter Name is missing (0xD0200)";
    Log.debug("Validator: " + msg, getClass().getName());
    return Result.err(msg);
  }
  return Result.ok(true);
}
```
Java

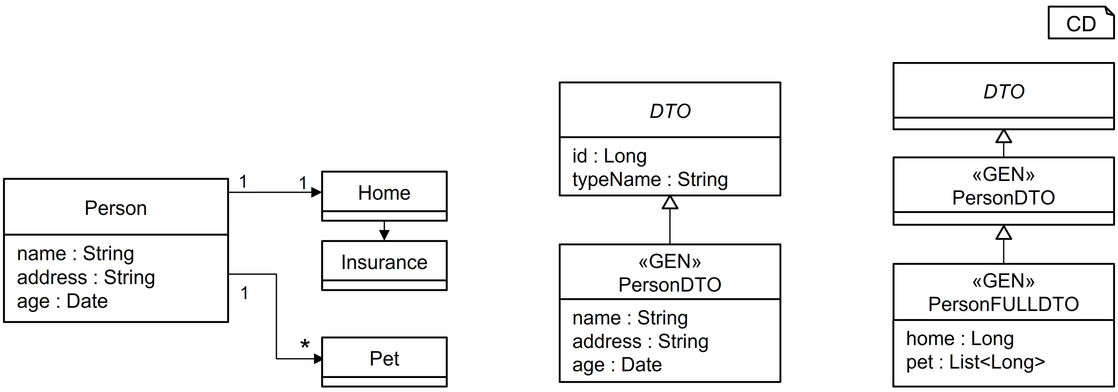
Listing 7.3: Generated validation method for an attribute, without additional validation defined by an OCL-Model.

Data Transfer Objects

The MontiGem architecture can be divided into two major components: The front end (Client) and the back end (Server), as shown in Figure 7.4 and Figure 7.6. In order to operate the web information system, data must be exchanged between client and server on a regular basis. Since loading and sending data is resource-intensive, it is necessary to send no more than the absolutely necessary data.

For this purpose, we divide objects that can be sent into three categories: Domain Classes, FullDTOs, and DTOs (**D**ata **T**ransfer **O**bject). We refer to Data objects that incorporate at least every association and attribute defined in the class diagram as *Domain Classes*. Beyond merely housing its attributes and directly associated objects, a domain class object also encompasses objects that are transitively linked via associations. This facilitates the transmission of an object tree (see 7.11(a)). While this constitutes the most resource-intensive data transfer method, it is sometimes indispensable. For instance, when establishing a new intricate object, it is frequently dispatched to the back end together with its associated objects. Web information systems often only load directly required data in order to reduce the initial load time. Thus related objects are often reduced to a unique ID, which allows us to load data from the server at a later time. This concept is implemented in MontiGem through the FullDTO (7.11(c)). It reduces all linked objects to either a `Long` attribute for associations without multiplicity or `List<Long>` for associations that can link to multiple objects. Often, especially when a specific object is in focus, but not its context (e.g. changing the name of a Person-Object, does not affect further objects), only the data object itself, but not its linked objects are needed. Therefore, the smaller DTO can be used. It only contains the attributes that are defined in the domain model, next to the unique id of the corresponding object in the database. This lightweight object is used to read, update or delete objects and is well-suited for loading many objects from the database. Next to the DTOs the generator also provides list-objects for both FullDTO and DTO (FullDTOList, DTOList) that ease the transmission of multiple DTOs within one object.

Up to this point, we know that the generator creates a database scheme, database access, and DTOs based on the same model. Thus we can generate a class that initializes DTOs with the corresponding data from the database if provided with a corresponding ID. Therefore for each class in the domain model, a `DTOLoader` is created. Listing 7.4 shows an example of a generated DTO loader initializing a PersonDTO. The method `loadDTO(long id)`(Line 1) is provided with the database id that is used by `DAO` (Figure 7.8) to retrieve the object from the database. The method throws a class-specific exception (Line 4) in case there is no object of the given type with the given ID in the database. MontiGem also provides a `loadDTO()` method that does not take an id as an argument. It returns all entries of the given type found in the database.



(a) Domain Class with Associations (b) DTO (c) FullDTO

Figure 7.11: (a): Domain Class with associations. (B): DTO, not containing associations. Instances of a DTO do not contain any information about linked objects of the corresponding domain object. (c):FullDTO, resolving direct associations with unique database IDs. There is no information stored about associations of associated objects (PersonFullDTO has no information about 'Insurance' object.)

Java

```
public PersonDTO loadDTO(long id) {
  Optional<Person> person = DAOLib.getPersonDAO().findAndLoad(id);
  if (!person.isPresent()) {
    throw new NoSuchElementException("Can not load " +
                                      "'Person' with id: " + id);
  }
  return new PersonDTO(person.get());
}
```

Listing 7.4: Generated DTO-loader method for Person-Class defined in Domain Model (*cf.* Figure 7.5). `loadDTO` will throw an error in case there is no Object persisted for the given ID. There are further generated methods such as `findAndLoad()` that can handle missing data and will return `Optional.empty()`.

Commands

MontiGem implements the *command pattern* [HH13] in order to manage communication between server and client. The DS-Generator provides for each class in the domain model commands to create, read, update, and delete objects in the database. Additional commands are generated to set single attributes of an object and to get lists of objects

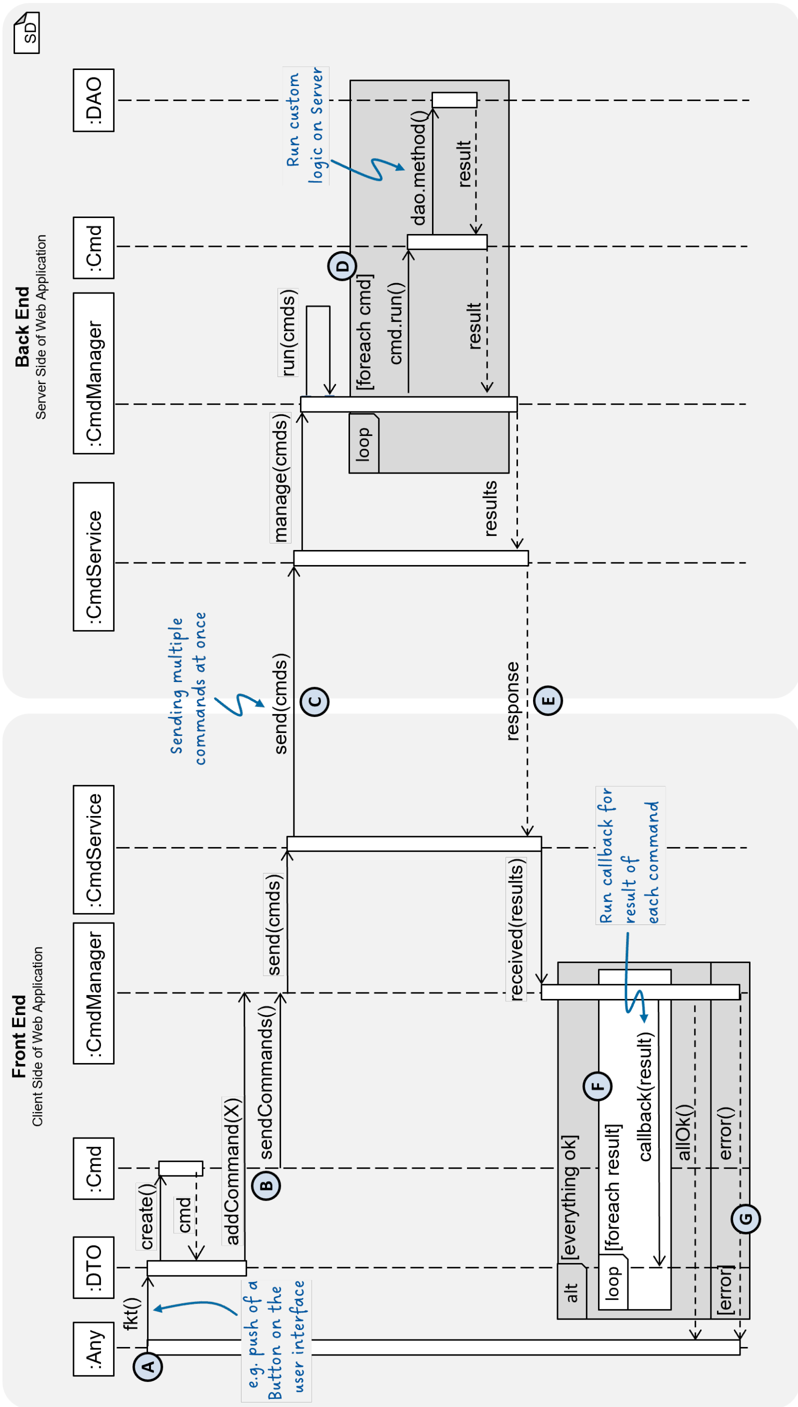


Figure 7.12: Sequence Diagram for the command usage in MontiGem. A command assembled in the client is sent via the command manager to the back end. The command is processed at the server before a response is sent to the client.

from the database. In Figure 7.12 a typical process of command execution is shown:

Ⓐ An event at the client triggers the creation of a command (e.g. initializing a page in the web application or reacting on the push of a button). At this point the `create()` method of the command manager is used to build a new command. At this point the callback function, defining the code that should be executed upon a response of the command, is defined as well. Once the command is created, it is added to the command manager.

Ⓑ After any number of commands have been added to the manager, they can be submitted via the `sendCommands()` method.

Ⓒ The commands are transmitted to the command manager of the server and delegated to the respective implementations of the commands in the back end. For every generated command implementation in the client, there is a matching generated implementation in the server.

Ⓓ Each command implementation on the server side implements a `doRun()` method, that itself executes three methods *cf.* Figure 7.13. Each command checks first if the received data is complete (`checkContract`) then if the current signed-in user has the required permissions to run the command (`checkPermission`) and finally runs a generated logic (`doRun`, Figure 7.12). The `doRun` method is typically overwritten to execute custom logic. Every command produces either the expected DTO, or an ErrorDTO containing a thrown error and a description of the problem.

Ⓔ The responses are gathered by the command manager and returned to the Client.

Ⓕ Upon receiving the results, the client runs each previously defined callback method for each received response.

Ⓖ In case of the reception of an ErrorDTO the client can react with a custom callback as well, e.g. by displaying an Error Message to the user.

7.3.2 View Model

As initially presented, MontiGem uses three groups of class diagrams as input: The first group is the domain model (as defined in the previous section Section 7.3.1), which serves as the core of the data structure and defines the database and the core domain-dependent infrastructure of MontiGem. The second group contains view models. A view model defines a view of the database. It can define aggregated, recomposed, or derived parameters that can be clustered in new classes. View models are used to define a data structure that fits a specific user interface and can be mapped to the data structure of the database. Thus, they define the objects that are only transferred between server and

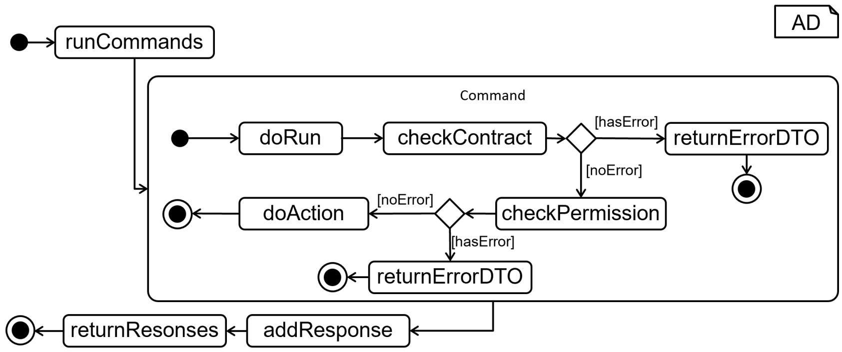


Figure 7.13: Every generated command at the server follows this standardized process: A command is executed via the `doRun()` method. `checkContract()` evaluates the well-formedness of the received data. `checkPermission()` evaluates whether the current user has permission to run this command. `doAction()` contains the logic that should be executed (e.g. CRUD Operations).

client; therefore, these objects are not persisted in the database. Therefore MontiGem generates for each class in a view model a DTOLoader class that governs the mapping from the domain model to the view model:

- **DTO:** (Described in Section 7.3.1) The same transformations are used to generate DTOs from View models as for the domain model.
- **DTOLoader:** A DTOLoader is generated as well. This DTOLoader however does not contain logic, that maps the data from the database to the DTO, as this logic is not directly derivable from the class diagram.
- **Commands:** (Described in Section 7.3.1) For each class in the view model a command to get a specific object (`getById`) and a command to get all objects of that type (`getAll`) is generated. Similar to the DTOLoader, commands that directly load the data from the database can not be generated. The generated commands need to be overwritten with custom logic by extending the generated DTOLoaders in order to work. A not extended command will return an `ErrorDTO` Containing the `NotYetImplemented`-Error.

In contrast to the domain model, the generator does neither generate database tables and corresponding DAOs nor Builder and corresponding Validators for the classes in the view model. Note that classes from the domain model can be referenced in the view model, thus reusing class definitions e.g. to define a specific payload of a command without having to redefine each class.

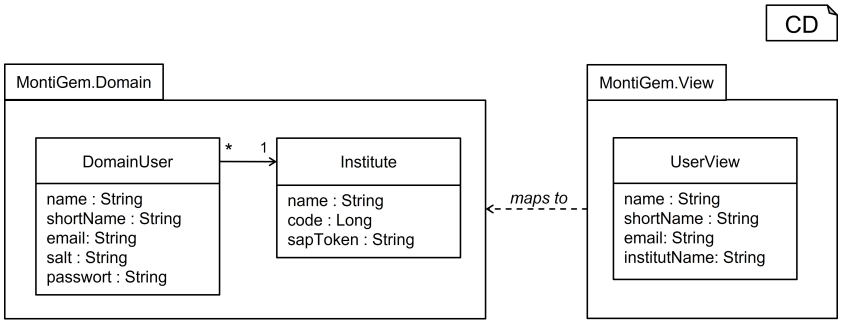


Figure 7.14: Class diagrams of View Model and Domain Model side by side. The parameters `name`, `shortName` and `email` map to `DomainUser`, The attribute `institutName` maps to `Institute`.

7.3.3 Command Model

MontiGem can be provided with additional models to generate commands only. These commands are used in case no data payload for the command is needed and the intent is only to trigger an event on the server. These commands are also used, in case there are multiple commands needed, that would use the same payload but serve different purposes, e.g. deleting an object in two modes: in the MaCoCo use case (*cf.* Chapter 8) the `forceDelete` command removes an object with all associated objects without warning, in contrast to the normal `delete` command, which informs the user of any problems. Both commands use the unique database id to target an object, but the command class diagram is needed to generate an additional command and the corresponding infrastructure to add the handwritten deletion logic. The command class diagram allows the developer to add additional commands to the command infrastructure without adding additional DTOs and DTOLoaders to the existing infrastructure. Note that both view models and domain model can be referenced in the command model thus reusing class definitions.

In summary, MontiGem generates for the three groups of class diagrams multiple components (As shown in Figure 7.15):

1. **Domain Model:** A complete command infrastructure from the database through the server to the client for the domain model.

2. **View Model:** Commands and DTOs to move data from the server to the client.

3. **Command Model:** Only Commands that reuse DTOs to move data.

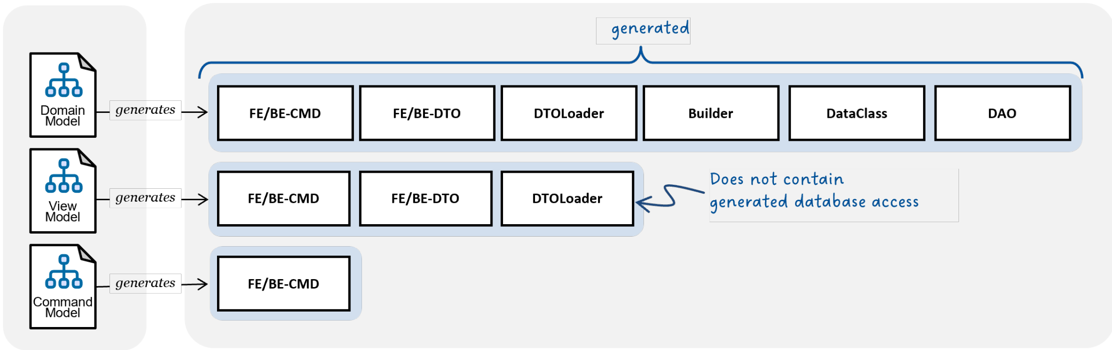


Figure 7.15: Not all Models are used to create the same amount of artifacts: Different components are generated for different groups of input models.

7.3.4 Constraint Model

Data consistency is an important issue for any information system. A mechanism is needed that ensures that any data sent or retrieved from the database is valid. Similar to the test case generation presented in Section 6.5.2, there are several layers to the validity of data. The first is a simple type check, ensuring that data is stored in the correct type and format. The second is a data structure consistency check. This ensures that objects and their relations match the underlying data structure of the application. Finally, a constraint-based consistency check, testing for the semantic validity of provided data.

Since the generator was implemented with Java (Strongly Typed GPL) we can take type checks as given. The client however is based on TypeScript and thus does not enforce type-checking natively, commands are serialized to JSON, a format that also does not enforce type correctness. A mismatch in command definition could still lead to a correctly typed object being serialized to JSON and wrongly de-serialized at the back end. Therefore, the command manager can handle type check errors and will inform the client accordingly.

The generated builder classes ensure validity based on the domain model. Before building, constraints set up by the class diagram are checked, such as cardinalities and obligatory attributes.

The third kind of check is the semantic validity: For example, ensuring that an `age` attribute is never larger than 200, or that every `username` has to be unique.

For each element in the domain model constraints can be defined in OCL syntax. These constraints are enforced in the back end by generated builders (Section 7.3.1) through corresponding generated validators and in the front end by generated validators that are available for user interface optimization (e.g. notifying that user input is invalid, before sending it to the server). Note that validation on the client is only an optimization to the user experience, data integrity is ensured by data validation in the back end. Any data received by the server will be validated in the back end before persisting it to the database. Validation at the client helps the user to find erroneous input without a noticeable impact on latency; this is especially helpful in case larger objects are created in the front end. Listing 7.5 shows an Constraint Model for the constraint `isAgeNotNegative` for the `age`-attribute of the Person-Class described in Figure 7.7. The constraint tests if the age parameter is positive (Line 3).In case the condition should fail, the model offers two texts for the output of error messages. In MontiGem these are used in the user interface to inform the user of any invalid input on the user interface directly (`shortError`) and to inform the developer by logging (`error`).

```
1 ocl ExampleConstraints {                                        OCL/P
2   context Person inv isAgeNotNegative:
3     age >= 0;
4     shortError: "Negative Age";
5     error:      "The age has to be a positive integer!";
6 }
```

Listing 7.5: Example of an OCL model for the `age` attribute of Figure 7.7. The Model contains next to the Constraint itself (Line 3) also Error messages (line 4,5) to display on the User Interface or in Error Logs.

Validation steps are generated separately for both front end and back end, as the server and client can be implemented in different programming languages [ANV+18]. The principle methodology, however, remains similar in both the front end and back end: Constraint models are parsed and checkers are generated in each corresponding language. These validators are either included in Builders (back end) or in forms (front end), where the including component can react on a violation of the check. In the back end, this would result in an error sent to the front end and a cancellation of the creation or update process of an object. In the front end, this would prevent the user from sending an invalid create or update command. Figure 7.16 shows the generation process for validators.
The resulting code for both server and client is shown in Listing 7.6 and Listing 7.7. It shows the check method for the constraint defined in Listing 7.5, which validates the age parameter of a Person-Class.

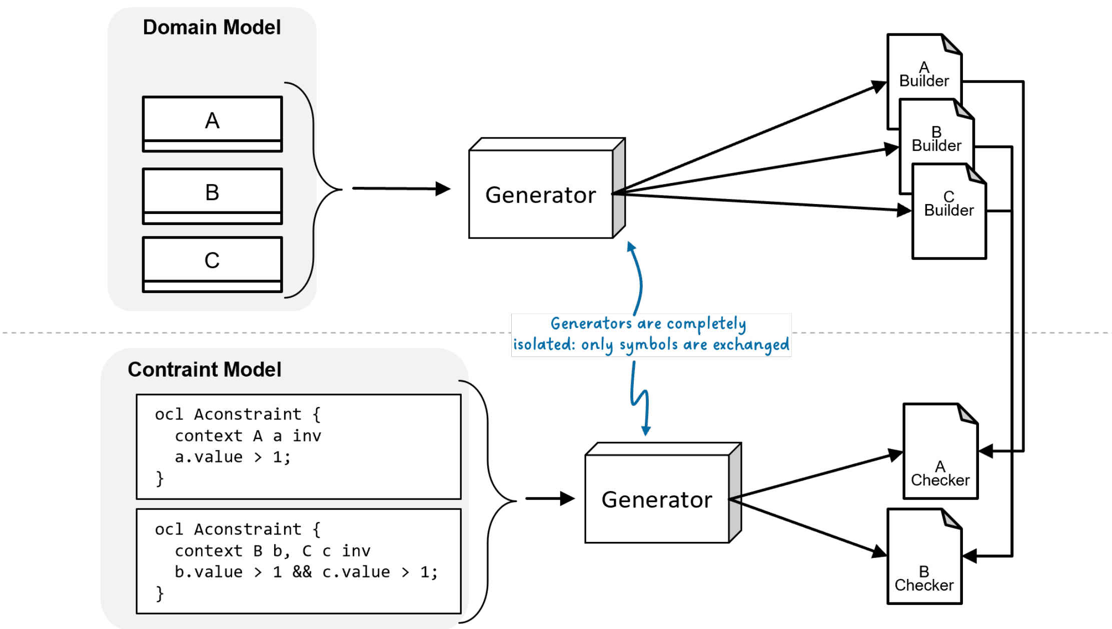


Figure 7.16: Builders and Checkers are generated by separate generators components.

```java
if (age < 0) {
    return age.getHumanName() + " is not correct \n";
}
```

Listing 7.6: Generated if-clause implementing the constraint check in the back end. The method returns a message that provides further information to the user ('*Age is not correct*') on the specific violation and can be sent to the front end, together with other error messages.

```ts
if (age !== null && age < 0) {
    throw new ValidationError("age is incorrect\n");
}
```

Listing 7.7: Generated if-clause implementing the constraint check in the front end.

7.3.5 Tagging

The complexity of languages chosen for defining input models is intentionally minimized. For CD4A (*cf.*: Section 3.4), we adhere to the UMLP specification [Rum17] of class diagrams. In the MontiGem-usecase however, a simple data structure definition is not sufficient, as we also need to know further information like persistence strategies, or database configurations. Our solution is to employ an auxiliary model, the tagging model (*cf.* Section 3.5), which supplements this extra information without altering the foundational language. An alternative to tagging is the usage of stereotypes. MontiGem is capable to process several stereotypes as discussed in Section 6.6.2.

The introduction of a tagging language entails the adaptation of the generator framework used. Although we have not yet explained the generator at this point, we briefly present the necessary adaptations to it here. Figure 7.17 shows the generic MontiCore-based architecture the MontiGem-framework adheres to. A complete description of the generator can be found in Chapter 7.

To incorporate tagging into a generative approach, we need not only to define the target language-specific tagging DSL but also to extend the generator that uses the tagged models. In the following example (*cf.* Figure 7.17), we consider tagging for CD4A. Once the DSL-specific tagging grammar (1) is defined (*cf.* Section 3.5.1), the developer can define a Tag Schema and corresponding Tags (*cf.* Section 5.3.1) for the input models (3). We can use MontiCore to provide a parser (2) that can handle both the CD4A model as well as the Tagging model. The resulting input AST thus contains both the CD4A model as well as the corresponding Tags. As the Tagging language extends the targeted DSL (c.f. Section 3.5), we can treat the AST as an CD4A AST where needed, minimizing the required adaptations in the generator when switching from an untagged to a tagged model. The model transformer can still use the CD4A-specific function library to transform the AST. Additionally, the developer can add tag-specific functions (5). A corresponding transformation to Listing 5.42 would be the addition of a `toString()` method to each class, that uses the `HumanName` from a tag instead of the attribute name itself. Finally, the extended generator will produce the target code (6), just like a regular generator.

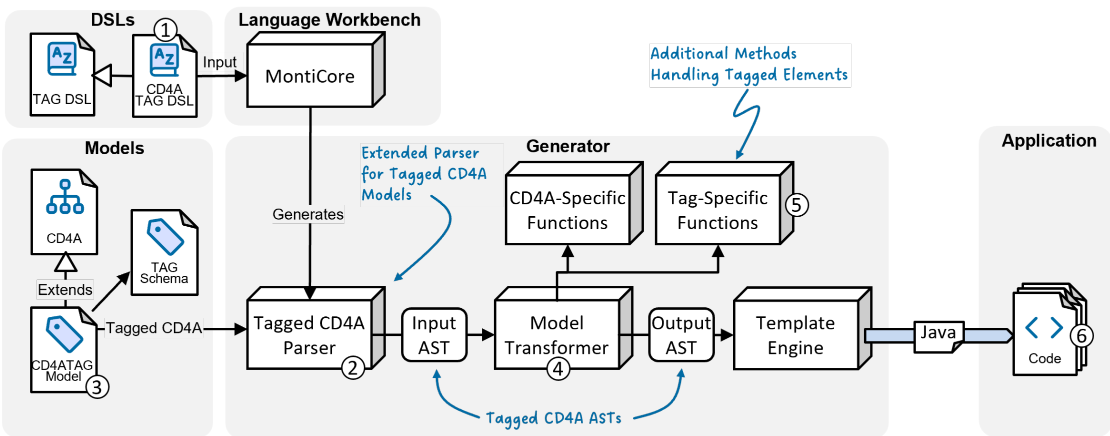


Figure 7.17: Extending a generator (As shown in Figure 3.2) for tagged models.

Within the MontiGem framework, tagging models are used to enrich AST-nodes in the domain model with further details. By default, MontiGem supports two tag varieties: database configurations and additional labels for user interface enhancement, such as 'Human Readable names' for both attributes and classes.

```
tagschema DBTags {                                          TagSchema
    tagtype NoCascade for Assoc;
    tagtype ColumnDef:String for Attr;
    tagtype UniqueDBColumn for Attr;
}
```

Listing 7.8: `DBTags.tagshema`: Tagschema defining what symbols in the class diagram can be tagged with which tags to configure database behavior.

Listing 7.8 shows the default `TagSchema` for database configurations within MontiGem. There are three tags defined: `NoCascade` for associations, `ColumnDef:String` and `UniqueDBColumn` for attributes. Adding DBTags (*cf.* Listing 7.8) to an attribute can configure the generator with `ColumDef:String` to create the database to store large amounts of text for this attribute and to ensure uniqueness of a field in a table for an attribute with `UniqueDBColumn`. The `NoCascade`-Tag configures the database to not cascade data operation over associations, e.g.removing one object will not cause the deletion of all connected objects.

```
tagschema HumanTags {                                       TagSchema
  tagtype HumanName:String for Attr, Class;
}
```

Listing 7.9: `HumanTags.tagshema`: Tagschema defining what symbols in the class diagram can be tagged with which tags to add another designation.

Another `TagSchema` supported by MontiGem is the `HumanTags`-Schema (*cf.* Listing 7.9). Attributes and Classes can be tagged with the `HumanName:String` tag. They are used to add another designation to either an attribute or a class. This additional information is especially useful for any automatically generated user notification or error message. Often the naming of classes is either very technical or in the wrong language (e.g. the system was developed in English, but the application language is German). The tags can be used to easily add an end user-friendly designation that should be shown in the user interface. An example is shown in Listing 7.10.

Tag

```
conforms to tagschema.HumanTags;
tag Account.startDate with HumanName = "Project start date";
tag Employment.emplType with HumanName="Type of employment";
```

Listing 7.10: Example for tags that improve the readability of specific attributes for the end user. Putting a generic `startDate` into a Project context and writing out abbreviations.

The generator can easily be configured and extended to support further tags. In several projects, tags were used as well to define user access and permissions on specific attributes (*cf.* Section 6.5.1) and classes or to define how test data should be created (*cf.* Section 6.5.2).

7.4 Generator for Graphical User Interfaces

The MontiGem-framework contains several generators to create a web application. In the previous section the generator that synthesizes the data structure and corresponding infrastructure for transmission and persistence of domain-specific data objects was described. In this section, we will take a closer look at the generator for the graphical user interface that synthesizes code that resides only in the presentation layer.
The web application generated with MontiGem is a single page application, thus the generator for the graphical user interface (short: *GUI-generator*) provides components that are embedded into a routing structure within the angular framework. This approach was chosen as it allows to dynamically rewrite the current web page with new data from the server instead of reloading the entire page. Different components are loaded based on the URL provided by user navigation, or entered by the user directly. In this context, we refer to a *page* as a component that is loaded upon a specific URL request, e.g. a dashboard or a settings page.

7.4.1 Generator-Architecture

The GUI-generator is based on the MontiWis [RR13, Rei16] and MontiWeb approach [DRRS09]. MontiGem uses models defined in the GUIDSL to implement the *Model-view-viewmodel* pattern [And12, HH13]. The implementation of both model (Section 7.3.1) and view model (Section 7.3.2) is provided by the data structure generator, leaving only the implementation of the view to the GUI generator.
The GUI-generator is provided with a set of GUI-models. Each model is parsed and processed individually and will be transformed into one corresponding page (*cf.* Figure 7.18). Once a model is parsed, it is provided to four transformers: (1) One for providing the TypeScript implementation representing the displayed angular component for a page. (2) One for the corresponding HTML implementation that references the component. (3) One for the form that might be required for the component and (4) a validator that is used by the form. Each transformer passes the output AST to a template engine, that generates the target code. The GUI-generator creates, next to navigation and the TypeScript compiler configuration `tsconfig.json`, the TypeScript and HTML implementation of a page. If specified in the GUI-model the generator provides a form and a validator for that form as part of the corresponding page.

Generating the TypeScript Component

In order to create a page, the generator uses a set of templates that are nested within each other. Figure 7.19 shows the template structure that is used to synthesize Typescript code for one page. Within the transformation step (TypeScript Trafo) the AST is extended by multiple methods that are derived from the GUI-model, e.g., each table defined in a gui-model will result in an additional init method for this table, also each

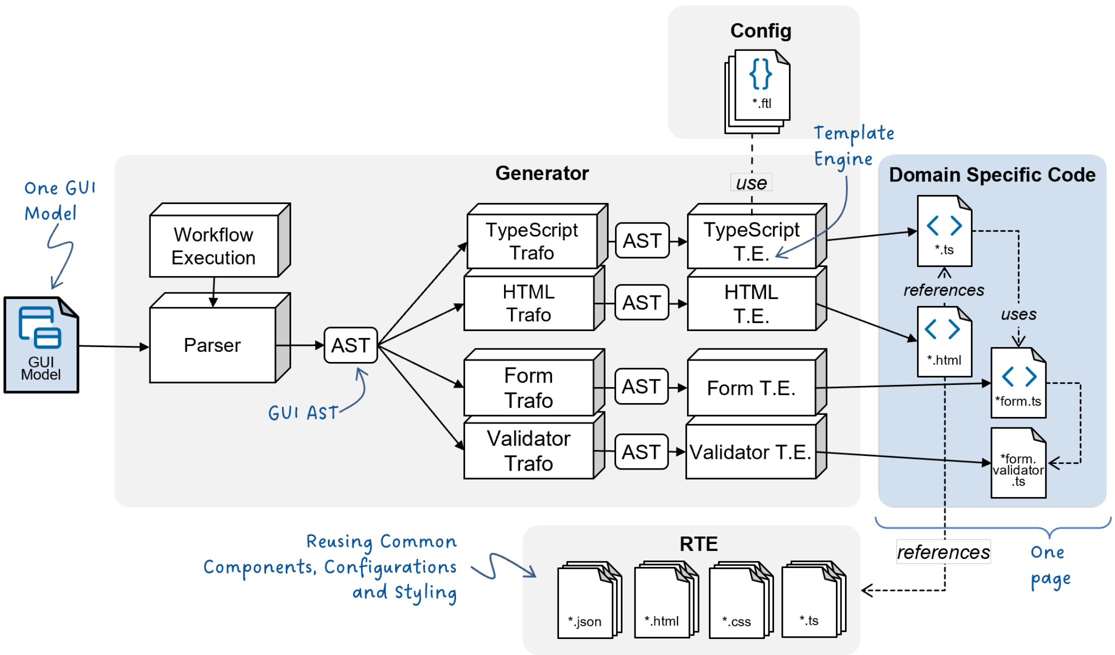


Figure 7.18: GUI-generator producing an Angular Component and a corresponding HTML file with two different template engines, that can be embedded in the single page application.

specified DTO will result in a method to gather the corresponding data from the server. Next to the model-dependent methods, there are default methods such as `NgOnInit` or `OnrouteChange` that are common to all websites in an Angular environment. Once AST is extended with methods, attributes, imports, and annotations, it is passed on to the Template Engine. Within the template engine, the AST is passed to the core-template `TSPage.ftl` that defines the overall structure of the targeted TypeScript file. Listing 7.11 shows the FreeMarker Template used to define the structure of a TypeScript file: In the first line, all the necessary AST nodes and parameters are provided to compose the targeted class in TypeScript: the name of the class, required imports, implemented interfaces, class attributes, constructor definition, and method definitions. In the next line, the template `Imports` is used to define the implementation for the import statements. The class body is defined from lines 3-14. In case the interface should be implemented, the provided classes are added in line 5 as a comma-separated list. Similarly lines 7-9 list the class attributes. The `Attribute`-Template is used for each attribute. Line 10 calls the `Constructor`-Template with the corresponding Constructor-AST-node. Lastly, all provided methods are listed by using the `Method`-Template with the corresponding AST-nodes.

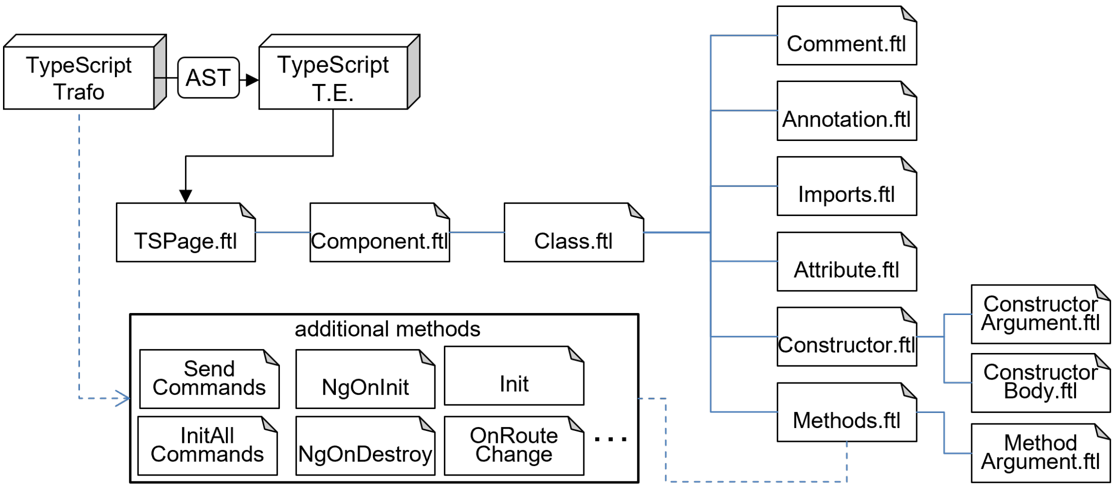


Figure 7.19: Template structure used in the GUI-generator to synthesize Typescript code for one website.

```
${tc.signature("className", "methods", "attribute",
    "constructor", "imports", "interfaces")}
${tc.includeArgs("coretemplates.Imports", [imports])}
export class ${className}
  <#if interfaces?size gt 0> implements
  <#list interfaces as interface>${interface}<#sep>,</#list></#if>
{
  <#list attributes as attribute>
    ${tc.includeArgs("coretemplates.Attribute", [attribute])}
  </#list>
    ${tc.includeArgs("coretemplates.Constructor", [constructor])}
  <#list methods as method>
    ${tc.includeArgs("coretemplates.Method", [method])}
  </#list>
}
```
FTL

Listing 7.11: Core template used in the GUI-generator that defines the basic structure of a TypeScript class: `Class.ftl`(Figure 7.19)

Generating the HTML Component

The HTML code generator is structured differently from the TypeScript generator, as the target programming language is a markup language instead of an object-oriented programming language. Instead of processing the AST-nodes in the stack (e.g. attributes first, methods next), the elements from the GUI-model can be processed in the sequence as they are defined in the model. The visitor pattern is used to set a specific template for each node, which is used later when synthesizing.

```
${tc.signature("ast")}
<#list ast.getPageElementList() as p>
  ${tc.include("gui.html.core.EmptyTemplate", p)}
</#list>
```

Listing 7.12: Core template used in the GUI-generator that defines the base of a HTML File.

Generating Navigation

In MontiGem also the navigation bar within the application is also based on a GUI-model (See Section 5.1.6). The corresponding GUI-model is provided separately from the other models and processed in its own generator process. The GUI-model does not define the routing or application modules, it only provides a structure for the user interface that provides the user with means to interact with the existing front-end structure.

Generating TSConfig

In order to set up the TypeScript project the configuration file `tsconfig.json` is needed. It defines the TypeScript-Compiler options and sets the multiple paths of both handwritten and generated artifacts within the project. The configuration file can be extended multiple times, thus there is one configuration that is provided with the application and a generated one that contains all domain-dependent configurations, extending the provided one. The generated one is synthesized through a simple template (`TSConfig.ftl`), configuring the list of needed paths to the generated artifacts in front end.

7.4.2 Overview of Generated Artefacts

Figure 7.2 depicts the artifacts that are generated by MontiGem for both the server and the client. In the shown use case, MontiGem is provided with a set of models (A): A domain models defining a `Person` class a view model `PersonDashboard` and two GUI-models for a web page `person.dashboard` and the navigation `main.navigation`. Based on the GUI-model for the web page, a corresponding TypeScript- and HTML file is generated (B). In addition, commands are generated (C)(*cf.* Section 7.3.1). In case the GUI-model is defined to load all available Objects of the `PersonDashboard` type, the `persondashboard.getAll.ts` command is used. Based on the input class diagrams, the same commands are generated for both the Server and the Client. Therefore, a command serialized and sent from the client can be deserialized and interpreted from the server (D). As the command `PersonDashboard_getAll` targets a type defined by the view model, which is not persisted in the database, we need a handwritten mapping that loads persisted data into the requested objects. MontiGem generates an empty class, that has to be extended via the TOP mechanism with a handwritten one (E)(*cf.* Section 7.3.2). The handwritten `DTOLoader` can use the generated Data Access Objects (*cf.* Section 7.3.1) in order to retrieve data from the database (F).

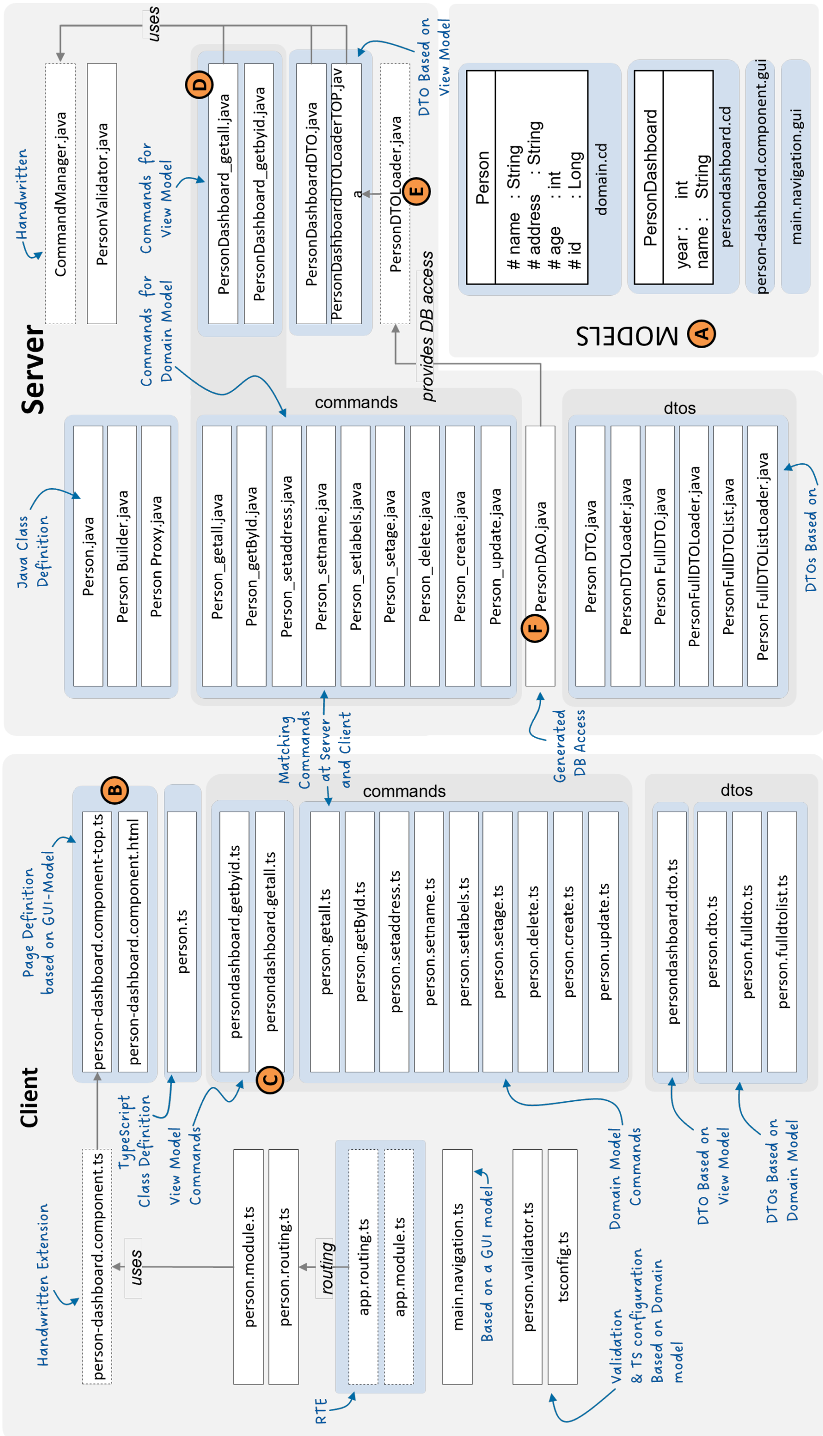


Figure 7.20: Artefacts generated by MontiGem for the class diagrams `Domain.cd`, `Roomdashboard.cd` and the GUI models `room-dashboard.componentn.gui` and `main.navigation.gui`.

7.5 Server Run Time Environment

The code generated by MontiGem is tailored to the corresponding architecture of the runtime environment. In the following, we will look at the architecture of both the client and server. The central components of the server are the server application itself: Apache TomEE, the persistence API Hibernate, and the Docker containers through which we deploy and manage the entire application. We will take a closer look at these in the following.

7.5.1 Used Server Architecture in MontiGem-Base Application

The server used as a back end is an Apache TomEE [Fou22]. Apache TomEE is an open-source, lightweight, Java Enterprise Edition (Java EE) application server that extends the capabilities of the popular Apache Tomcat web server. Apache TomEE is developed and maintained by the Apache Software Foundation.
Apache Tomcat, which is the foundation of Apache TomEE, is a widely used web server and servlet container designed to serve Java web applications. It was chosen for its performance, reliability, and ease of use. However, Tomcat only provides a limited set of features required for Java web applications and lacks full support for the Java EE specifications.
Apache TomEE bridges this gap by integrating additional enterprise-class components and libraries, such as the Java Persistence API (JPA) that we use with Hibernate to manage our persistence layer, Enterprise JavaBeans (EJB) that we use to manage DAOs, and Contexts and Dependency Injection (CDI) which we use among other implementations for our access control. These features enable developers to build and deploy robust, and secure enterprise applications using the Java EE standards.
In summary, Apache TomEE is used for:

1. Developing and deploying Java EE web applications and services.
2. Providing a lightweight, flexible, and production-ready application server for Java EE projects.
3. Extending the capabilities of Apache Tomcat with additional enterprise features.

Apache was chosen over heavier application servers like WildFly[9], GlassFish[10] or WebSphere[11] for the following reasons:
Apache has a proven track record of reliability, having been widely used for decades as a web server. Its long history and extensive community support ensure that any issues

[9] https://www.wildfly.org/
[10] https://javaee.github.io/glassfish/
[11] https://www.ibm.com/products/websphere-application-server

encountered can be quickly addressed and resolved. Furthermore, Apache is known for its high performance and ability to handle a large number of concurrent connections, which makes it suitable for EIS applications that require substantial traffic handling.
The flexibility of Apache is another significant factor in its suitability for building an EIS back-end. Apache is highly configurable and extensible through modules, which allows for customization to meet specific business requirements. This flexibility enables organizations to tailor their server configurations to the unique needs of their enterprise systems.
Additionally, Apache provides robust security features to protect sensitive enterprise data. It supports various authentication and authorization mechanisms, as well as SSL/TLS encryption for secure data transmission. The active development and maintenance of the software by the Apache Software Foundation ensure that security updates are continually provided, keeping the server up to date with the latest security practices.

7.5.2 Persistence Management in MontiGem-Based Applications

A core element of information systems is the simultaneous writing and reading of a shared database. For this we need a management layer that detects and resolves conflicts between read and write operations. For the MontiGem architecture, we have chosen the Hibernate framework.
Hibernate [Kin21] is an open-source Object-Relational Mapping (ORM) framework for Java applications. It provides a powerful and flexible persistence API for managing the persistence and retrieval of Java objects to and from relational databases. The core functionality of Hibernate revolves around mapping Java objects (also known as entities) to database tables and converting between object-oriented and relational data types.
Hibernate was chosen for the following reasons:

1. **Simplified database interaction:** Hibernate abstracts the low-level details of database interaction, allowing developers to focus on the business logic without worrying about writing complex SQL queries for basic CRUD (Create, Read, Update, and Delete) operations. Within MontiGem we use generated DAOs (*cf.* Section 7.3.1) to provide these operations to the developer.

2. **Database independence:** Hibernate provides a level of abstraction that makes it easier to switch between different relational database management systems (RDBMS) without having to change the application code. Although there is the option to switch RDBMS the major versions of MontiGem were only developed with MySQL and PostgreSQL. There have been theses evaluating MontiGem with other DBMS such as Neo4J and GraphDB.

3. **Improved performance:** Hibernate offers various performance optimization techniques like caching, lazy loading, and batch fetching, which can significantly im-

prove the performance of data access operations. Especially in later full-size real-world applications such as MaCoCo these optimizations yielded high-performance improvements. However, the generator had to be updated to make use of these features. Newer architectures of the generator make heavy use of lazy loading and batch fetching.

4. **Extensibility and customization:** Hibernate allows developers to extend and customize its functionality to suit their application requirements. For example, you can define custom data types or provide custom SQL queries for specific operations. Within MontiGem we use custom data types in order to optimize loading times and to load data directly into a Dataclass-DTO (*cf.* Section 7.3.1) if possible.

5. **Transaction management:** Hibernate integrates with Java transaction APIs (such as JTA) to provide a robust and consistent transaction management system for applications. Within the generated DAOs we use these transactions in order to prevent conflicts and inconsistencies within the database.

We use Hibernate to simplify data persistence in their Java applications, reduce boilerplate code, and improve overall maintainability and scalability. Hibernate was chosen as a persistence API in the early stages of MontiGem-development. A few experiments with further APIs were performed but did not indicate any advantage that would justify a switch away from Hibernate.

7.5.3 Deploying MontiGem-Based Applications with Docker

Docker [Hyk23] is an open-source platform that simplifies the process of developing, packaging, and deploying applications using lightweight, portable containers. These containers ensure that an application runs consistently across different environments by bundling the application along with its dependencies, libraries, and runtime environment. Docker addresses the "it works on my machine" problem, which occurs when an application works well in a developer's local environment but faces issues when deployed elsewhere. By providing a consistent environment, Docker reduces the chances of encountering configuration or dependency-related issues.
As we try to maximise to potential number of use cases and MontiGem is expected to be deployed in a large variety of environments. Docker was chosen as a deployment platform.
Here's an overview of the key components of a Docker setup:

- **Docker images:** A Docker image is a template that contains the application, its dependencies, runtime environment, and other required configurations. Images are built from a set of instructions written in a Dockerfile. Docker images can be stored in a public or private registry, like Docker Hub, for easy sharing and reuse. MontiGem uses four docker images as shown in Figure 7.21.

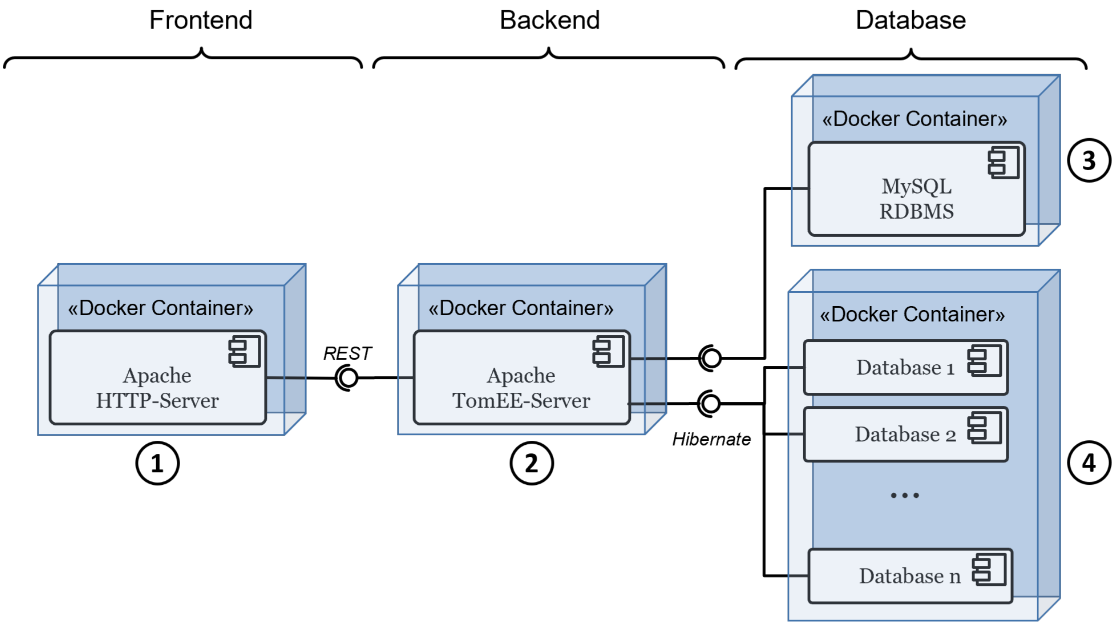


Figure 7.21: Docker Containers used in a typical MontiGem setup.

- **Dockerfile:** A Dockerfile is a script that contains instructions to build a Docker image. It specifies the base image, application code, dependencies, environment variables, and other configurations required to create the image. Developers can write a Dockerfile to define the desired state of their application environment.
- **Docker engine:** The Docker engine is the core component responsible for creating, running, and managing containers. It's installed on the host machine and communicates with the Docker daemon, which executes the commands and manages container life cycle operations. In order to deploy a MontiGem-based application the docker engine is the only element that needs to be installed, all remaining dependencies are provided within the container.
- **Docker containers:** A Docker container is a running instance of a Docker image. When a user runs an image, the Docker engine creates a container from that image, providing an isolated environment for the application. Containers can be started, stopped, and removed using Docker commands. MontiGem uses this aspect to redeploy parts of the application while keeping the persistence layer running. Individual components such as the Client can be updated without stopping or restarting the Server.

In summary, Docker works by using containerization to bundle an application and its dependencies into a portable unit. It provides tools and features to manage the entire

container life cycle, from building and storing images to running and scaling containers. This makes it easier to develop, deploy, and maintain applications consistently across various environments.
Figure 7.21 shows the container used in a Deployed MontiGem-Application. There are four kinds of containers. A container for the Apache HTTP Server (1), a container for the back-end server (2), and two containers for databases. One contains a generic database (3) that stores data relevant to the complete system and (4) contains databases for individual users or groups of users (instances) in which data specific to that users are stored.

7.5.4 Command API

MontiGem implements the command pattern [HH13]. Instead of implementing one API for every command it can receive, MontiGem implements one API that handles any command. A manager (`CommandManager`) handles the incoming commands and delegates to the respective command classes that process the input.

Services

The command service is an interface to external applications and the main means of communication for clients. The application back end is waiting for new commands and executes them. MontiGem implements two types of services, the *CommandRESTService* and the *WebSocketService*.
Both Services have the following in common (*cf.* Figure 7.12):

1. Each service receives JSON formatted string messages, which are expected to be serialized commands (list of commands),
2. The service attempts to deserialize and validate the commands, upon failure a error response (ErrorDTO) is returned to the sender.
3. The commands are passed on to the command manager which itself delegates to the corresponding command classes
4. Results from the command classes are returned to the Command manager which itself returns the answer back to the client.

In the event of an error, there can be different resolution strategies depending on the connection logic.
The **CommandRESTService** provides one endpoint for all commands. Commands received on this endpoint are collected by the command manager. The commands are processed sequentially. The results of the executed commands are collected and sent back to the client. Finally, the connection is closed. The server does not keep an open

session with the clients. It only receives the command and executes it according to its access control and the implemented business logic.
A REST connection always exists between one client and the server. If a command encounters an error, like when a necessary object isn't found in the database, then none of the subsequent commands in that list will run. Instead, an error message will be sent back. This approach is used because commands often rely on one another. If one fails, the next might not work properly.
The **WebSocketService** allows for a multitude of different executors, one of them is the CommandManager to execute commands. A WebSocket connection is opened between one client and the server. The client has to open the connection and provides a `messageType` which defines what kind of messages should be processed for this connection. The connection can be held open as long as the client or server need to communicate and is not closed upon a server response by default. With WebSockets, the server can also initiate messages to a client, if the connection is still open. In a REST connection, the server only can respond. As the connection is kept with exactly one client, the server knows which client has which connections and can send a message to specific clients. This asynchronous communication allows for the use of paging (sending data in iterations instead of one large transmission) in the execution of commands and enables the broadcasting of commands, e.g. for notifications from one user to another. This is useful if a certain event occurs and requires the attention of multiple users.

7.5.5 Security

As MontiGem is used not only in a test-case and proof-of-concept environment (research software with TRL 3) but also in industry and real-world scenarios (published software with TRL 9), we have to take into account the security of the generated application. The implemented access control (*cf.* Section 6.5.1) must be supported by an architecture that enforces the rules set by the access policy and ensures data privacy for sensitive data. AS MontiGem has been deployed in multiple projects [BGK+23b] containing sensitive data, the system has been improved over time to meet those security aspects. The following aspects were considered when developing MontiGem:
Secure Design: Security was considered from the beginning of the software development process. Designing the generator to produce target code with security in mind helped to prevent vulnerabilities and establish a generated secure foundation for the software. Permission checks, database restrictions, and user session management are part of the generated code.
Secure Coding Practices: Developers must follow secure coding guidelines and best practices to prevent vulnerabilities such as buffer overflows, SQL injection, and cross-site scripting (XSS). This includes proper input validation, output encoding, and error handling. As large parts are generated, the developers tend to follow the secure predefined patterns provided by the generator [GMN+20].

Authentication and Authorization: Implementing robust authentication and authorization mechanisms helps ensure that only authorized users can access specific resources or perform certain actions within the software. The used back-end server Apache TomEE provides both Authentication and Authorization as mentioned above, within MontiGem we use the Shiro[12] framework in order to manage access control for individual users.
Data Protection: Ensuring that sensitive data is protected both in transit and at rest is crucial. This can be achieved through encryption, hashing, and secure storage techniques. Within MontiGem, databases are encrypted and common security measures such as password encryption are implemented, as the architecture had to be validated by multiple data protection officers in order to comply with German data protection regulations.
Monitoring and Logging: Implementing monitoring and logging capabilities help to detect and respond to security incidents more efficiently. This includes tracking user activities, system events, and potential attacks. Although MontiGem supports in-depth logging, the developer has to ensure that corresponding monitoring tools are configured. Real-world applications generated with MontiGem used Icinga to monitor performance, availability, and unusual behavior.
Secure Deployment: Ensuring the secure deployment of software includes following best practices in server hardening, network configuration, and access control. Similarly to monitoring, MontiGem does not provide a deployment mechanism, the security of this aspect is up to the developer who configures the deployment for the specific use case. In multiple projects, MontiGem was deployed either via Jenkins or with GitLab runners.

Using Shiro in Generated Information Systems

Most of the aspects defined above are realized with the Shiro framework. The Apache Shiro framework, often just called Shiro, is a free security tool for Java applications. Its primary goal is to help developers manage various security elements, such as user authentication, permissions, session tracking, and encryption. Because of its versatility, Shiro fits seamlessly into a variety of platforms — from web to mobile apps, and even standalone environments.
Shiro was chosen for its following capabilities:
Authentication: Shiro boasts a dynamic authentication system. Developers can easily confirm user identities by comparing their given credentials, such as usernames and passwords, with a set data source. This could be a database or an LDAP server.
Authorization: With Shiro, it is simpler to oversee which parts of the application users can access. This is done by checking user roles, permissions, or other attributes, ensuring that resources are accessible only to authorized users.
Session Management: Shiro has an extensive session management feature. It is adaptable to various application types, supporting both stateful and stateless sessions. In addition,

[12] https://shiro.apache.org/

its transparent clustering is perfect for applications that need to scale.
Cryptography: Shiro has native support for many cryptographic tasks, covering hashing, encryption, and decryption. It is crucial for the secure storage of delicate data and safeguarding app communications.
Integration: Incorporating Shiro with other Java tools and frameworks is straightforward. It is compatible with many, including Spring, Java EE, and different web app servers.

Permission Management in MontiGem

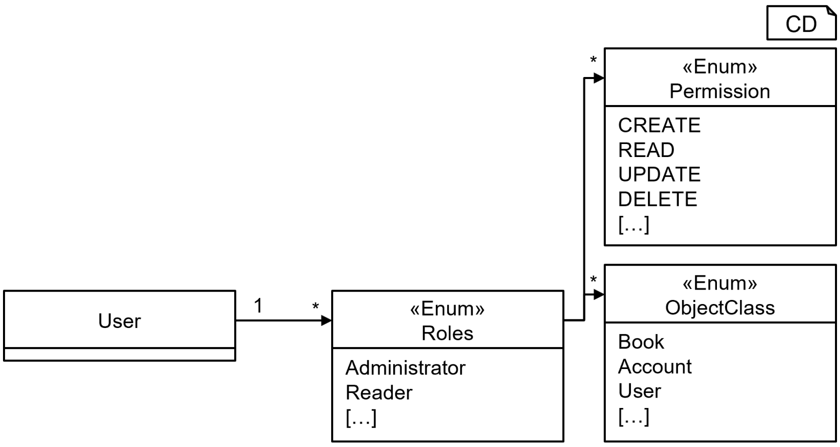


Figure 7.22: Data structure managing the permissions in MontiGem

The permission management and access control in MontiGem is based on the Shiro architecture: A user can be assigned to one or multiple *roles*. A role can have one or more *Permissions* and have one or more *ObjectClasses*. Thus, a user is granted a Role that gives him Permissions for specific ObjectClasses as shown in Figure 7.22.

$$Role \times Operation \times Class$$

The enumeration Permission defines the operation that we wish to control; these are the typical CRUD operations but can be also extended to control more specific functions, such as a data import or a conversion of an object. Although almost all operations can be reduced to CRUD, having additional operations is useful to enable differentiation between specific operations, e.g. it might be necessary to permit a user to change one parameter `x` of an object but deny him changing any other parameter, both operations would be an 'UPDATE', but by introducing a specific permission `changeX` we can grant a Role for this use case. The ObjectClass represents one or many classes that can be targeted by roles. Typically a Role is granted to perform an operation on a group of

classes: An account might have a budget, and that budget can have multiple bookings. It is infeasible to define permissions for each class directly, and thus we cluster all related classes. In case we want to check if a user has permission to update a booking, we check if he has permission to update the ObjectClass `Account`.
Let us take a look at an example. Within MontiGem we could define the following role: `AccountReader`: The Role contains the permission `READ` and the ObjectClass `Account`. If an account is loaded from the database, we can check if the current user has the read-permission for accounts, if this is the case, for example, because the user was granted the role `AccountReader`, then we proceed with sending the data, otherwise we can send an error or a notification.

7.6 Client Run Time Environment

In the previous Section, we took a look at the RTE of the server, in this Section, we will present the RTE of the client.

7.6.1 Apache HTTP

Apache HTTP Server[13], commonly referred to as Apache, is a widely used open-source web server software that was first released in 1995. It is developed and maintained by the Apache Software Foundation (ASF). Apache HTTP Server is designed to serve static and dynamic web content over HTTP and HTTPS protocols, and can be highly customized through a modular architecture that allows the addition of various features and extensions.
The primary difference between Apache HTTP Server and Apache TomEE lies in their purpose and functionality: Apache HTTP Server is a general-purpose web server, designed for serving static and dynamic web content. It is not inherently tied to any specific programming language or framework, and it can be used with various server-side technologies like PHP, Python, or Perl. Apache TomEE is specifically built for Java EE applications, providing a server environment that integrates a variety of Java-based technologies and APIs. This makes it more suitable for Java developers who need a lightweight Java EE server for their applications. Within the MontiGem-framework the TomEE Server is used to set up the back end and manage the business logic of the Application (Application layer *cf.* Figure 7.3). The HTTP server is used to provide the HTML and TS files to the client that are interpreted by the end-user browser (Presentation layer *cf.* Figure 7.3).

7.6.2 Usage of Angular in the MontiGem Client

Angular[14] is a popular open source JavaScript-based framework developed and maintained by Google, designed to build modern, scalable, and dynamic web applications. Angular is particularly useful for developing single-page applications (SPAs), where users can interact with the application without needing to reload the entire page. The framework has evolved over the years, with AngularJS being the first version, followed by Angular 2+ (referred to simply as Angular).
Angular was chosen for its component-based architecture, making it easy to structure and maintain code by dividing application logic into reusable and modular components. Each component consists of a template (HTML), a class (TypeScript), and metadata (annotations) which define the behavior and appearance of the component.

[13]https://httpd.apache.org/
[14]https://angular.io/

The code we generated is mapped to Angular. Therefore, we use some of its key artifacts to generate dynamic target code:

- **Modules:** Angular modules, also known as NgModules, are used to group related components, directives, and services together. They provide a way to organize and encapsulate code, making it easier to maintain and reuse. The root module, called AppModule, is the entry point for an Angular application, and additional feature modules can be created to organize related functionality. MontiGem generates one module for each page.
- **Components:** Components are the building blocks of an Angular application, responsible for defining a part of the user interface and managing its behavior. MontiGem uses predefined components as page elements that are displayed in the UI.
- **Directives:** Directives are used to add behavior to HTML elements or manipulate the DOM without writing JavaScript directly. There are three types of directives in Angular: component directives (components), attribute directives (to change the appearance or behavior of an element), and structural directives (to modify the DOM structure by adding or removing elements). MontiGem uses directives to hide and show elements in the UI via conditions (See Section 5.1)
- **Services:** Services are used to encapsulate reusable logic and data that can be shared across multiple components. They are typically implemented as classes with a specific purpose and can be injected into components using Angular's dependency injection system. MontiGem services are used to implement the command API within the client.
- **Routing:** The Angular Router is a powerful routing library that enables navigation between different views and components within an application while maintaining a clean URL structure. The routing configuration is usually defined in a separate module called AppRoutingModule. MontiGem uses the Angular router to navigate between the generated pages.

7.7 Modifying Generated Code

A distinctive feature of MontiCore is its embrace of the TOP-Mechanism [HKR21], which facilitates the composition and reuse of language components, as well as the extension and adaptation of generated source code. Through the TOP mechanism, the generated source code can be refined step by step.
A MontiCore-based generator such as MontiGem supports the TOP-Mechanism for any target language that supports inheritance. A class, for example, `MyClass` generated by MontiGem can be modified by the developer by adding the same class `MyClass` to the

handwritten code directory. MontiGem will detect the class as already present and generate a class with the new name `MyClassTOP`. The developer can extend the new class and thus overwrite any aspect he wants to modify.
A great benefit of this approach is its robustness against model changes. Modifying the generator to generate additional attributes, methods, or classes will not undo the changes of the developer as they only target specific elements of the class and do not replace the class as a whole.
Note that the TOP-Mechanism can not be applied to the HTML code and GUI-models as both languages do not support inheritance. The modification of GUI-models is discussed in Section 6.7.

Chapter 8

Case Study: Management Cockpit for Controlling: MaCoCo

The methodology presented in Chapter 2 targets not only the development of prototypes and research software (TRL 3), but also aims for an architecture that can transition into a full-size real-world system (TRL 9). This chapter serves as a demonstration that MontiGem is capable of producing an application that can transition to a full-size real-world system (*cf.* RQ1.3: *How to allow developers to incrementally transform the application from a prototype to a full-sized real-world system ?*).

Contents

Model-driven development of real-world systems opens up a new multitude of challenges [Sel03]. The software has to adhere to much higher standards in performance, user experience, and stability. Whereas research software often targets the optimization and analysis of one very specific aspect, real-world applications have to be sophisticated in a large variety of features. Use cases can not be implemented at a high level but rather have to cover every interaction a user might perform, and have to cover any errors that might occur. This leads to a tremendously larger software complexity and challenges the developer of the application. As a consequence, it challenges the developer of the generator framework as well.

In the following, we will introduce the MaCoCo use case. MaCoCo is a web application that was developed using the MontiGem-framework. At the time of writing, the domain model for MaCoCo defines around 150 classes. Additional 600 classes define DTOs and commands. 60 GUI-models define the user interfaces of the application, resulting in about 1.000.000 lines of code (*cf.* Table 8.1). The application manages more than 200 databases and serves about 200 users per day (*cf.* Figure 8.1). This proves that the approach is capable of transitioning from a prototype to a full-size application with an active user base.

Language	**Type**	**Lines of Code**
Java	Generated	600454
Java	Handwritten	204037
HTML	Generated	45468
HTML	Handwritten	7181
TypeScript	Generated	171408
TypeScript	Handwritten	76166
CD4A	Handwritten	4471
GUIDSL v1	Handwritten	12922
Total w/o Models		**1104714**

Table 8.1: Lines of Code of MaCoCo in Different Programming Languages

8.1 Motivation for MaCoCo

Similar to other enterprises Universities are working hard to modernize their management and operations through digital transformation [NPAB22]. They are paying particular attention to updating teaching, research, funding from external sources, and administrative work. Additionally, German universities became more independent as the government shifted the financial management to them, in order to decentralize administration and improve efficiency. As a result, universities established well-working accounting and reporting systems. The decentralization leads to new challenges in the context of digitalization: The central administration of each university is dependent on obtaining aggregated financial data from all its faculties, chairs and research institutes. The administration needs to supply financial reports, which include both the profit and loss statement and the balance sheet. Most central university administrations tackle these challenges by using a reliable well-established Enterprise Resource Planning Sys-

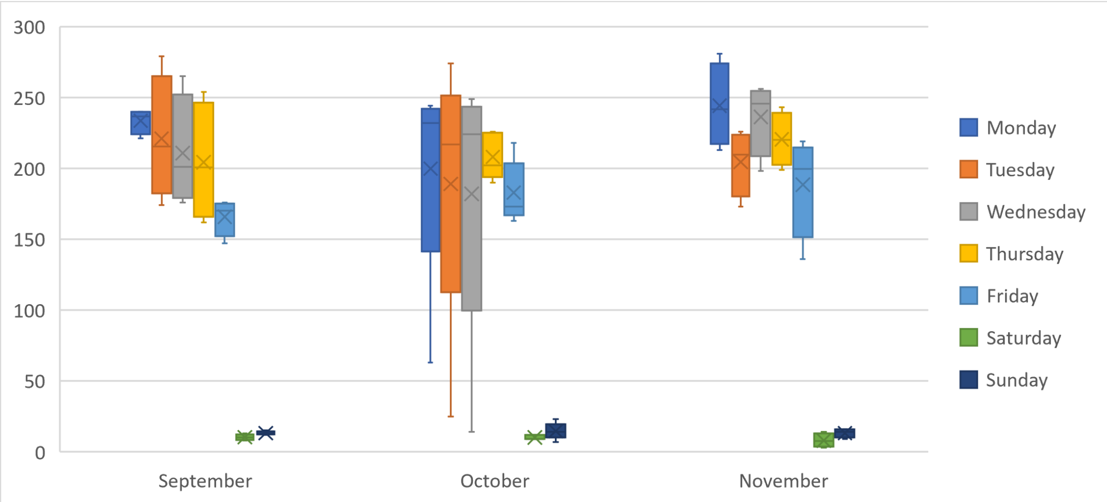


Figure 8.1: MaCoCo usage 2023. Amounts of Logins per day. Data shows a mean value of about 200 logins each workday with a strong decline during the weekend.

tem (ERP), such as SAP[1]. The controlling of chairs and research institutes on the other hand operate on a different scale. They are developing plans for financial resources, provided by the administration or third-party funding. The processes surrounding the spending of financial resources, are heavily regulated and follow numerous guidelines. Although there are sophisticated software solutions to handle the administrative tasks of the university administration at the university level, there are little to no tools that assist the chairs and research institutes to cooperate with the administration in their management tasks. Thus chairs and institutes have few options: (1) Use commercial accounting software, that is not adapted to the specific needs of the university domain. (2) Use spreadsheet software such as Excel or open office, that require the user to develop the calculations and manage the data themselves. These sheets are error prone as they are easily modified and do lack classical testing and validation. (3) Develop their own software solution for financial management. This option is only suitable for large chairs, as the software must be continuously maintained at high cost.

8.2 The MaCoCo Use Case

We take a closer look at the gaps and challenges presented considering the use case of RWTH Aachen University. The university was founded in 1870 and consists of 9 faculties. As of 2023, there are RWTH Aachen University 47,269 (Winter Semester 21/22)[2]

[1] https://www.sap.com/
[2] www.rwth-aachen.de/cms/root/Die-RWTH/Profil/ enw/Daten-Fakten/?lidx=1

students enrolled into one of 170 study programs. The university has 260 institutes and employs 553 professors. The institute sizes range from a few employees (<10) to more than 1000 such as at the WZL (*Werkzeugmaschienenlabor*).
Launched in 2016, the Management Cockpit for Management and Controlling project was set out to create a comprehensive enterprise information system tailored for RWTH Aachen University's chairs. This system was developed to aid in the planning, reviewing, and overseeing of management processes and cost accounting, streamlining these critical operations for efficiency and effectiveness. The project is run by a collaboration between two chairs at RWTH Aachen University: the Chair of Management Accounting[3] from the Faculty of Economics, and the Chair of Software Engineering[4] from the Department of Informatics.
In its initial project definition, MaCoCo was aimed at small and medium-sized chairs of the university that would encompass a small organizational structure with few workflows and simple hierarchies. The MaCoCo project initially focused on the smaller to medium-sized chairs, however as the project matured larger entities such as subdivisions of WZL expressed interest in also using the software. Thus, during the development, larger institutes of the university showed interest and MaCoCo was adapted accordingly. Large chairs possess more extensive administrative structures and are more oriented towards specific workflows. These chairs have distinct needs concerning their financial matters and often employ systems akin to those used by private-sector companies for accounting and sometimes for workflows. Although the integration of large departments presented the project with several new challenges that were difficult to solve, the smaller departments were also able to benefit from the new features. Runtime optimizations and professionalization of the processes in the system benefited all users.
MaCoCo serves as a great platform to develop and test model-driven methodologies. Especially aspects such as the generator's adaptability to changing requirements, its robustness, and scalability. Additionally, the MaCoCo use case provides a highly detailed scenario the preceding transformers can be implemented for. For example: Both CD2GUI and LLM4CD have to be able to handle models as large as used in MaCoCo and the GUI-models produced by CD2GUI must be adaptable enough to match the complexity of the MaCoCo's GUI-models.

User Groups

One of the objectives of this thesis is to use model-driven software engineering to bridge the gap between developers and domain experts. Following this principle, three user groups were formed to be included in the software engineering process:
Lead Users. Lead users are domain experts of and represent the interests of the average user of the finished product. The development of a complex system such as MaCoCo

[3]Chair of Management Accounting: `http://www.controlling.rwth-aachen.de/go/id/mgaz`
[4]Chair of Software Engineering's project page: `http://www.se-rwth.de/projects/MaCoCo.php`

requires a great deal of expertise in the areas in which the platform is to be used. As part of the agile development process, potential end users are consulted to develop and optimize the application iteratively. In order to streamline this process, only a few individuals are chosen as lead users, representing the group of end users as a whole.

The principle idea behind faculties within a university is to group and separate chairs based on their research domain. However, this separation also entails differences in the administrative processes. To cover this diversity the group of users are chosen from all eight faculties. In addition, the users were picked to cover small and medium-sized chairs and deaneries as well.

Steering Committee. As MaCoCo offers functionalities that target processes that handle sensitive data such as staff management, or financial management, the development team relies on the guidance and oversight of the steering committee to produce a correctly regulated software product. Each member of the steering committee monitors a specific aspect of the development process of the application, as they represent rules, values, or interests of a group of people. The members consist of the university chancellor's representative and the head of the department for finances, representing the universities' interests in their investment. The two personnel councils and the department head for personnel represent the rights and values of the employees. Together with the data protection officer, we ensure the collected personnel data is secure and evaluate potential analysis and presentation methods for the data. MaCoCo is hosted by the university IT Center and interfaces with the SAP ERP system for university administration. Consequently, representatives from both the ITC and the SARA project are members of the steering committee. In addition, the committee includes the head of the "Organization and IT" department, the dean and managing director of the faculty for mathematics, physics, computer science, and natural science, as well as a representative and a professor from various computer science chairs. In addition, to represent the interests of the lead users, two of their representatives attend the biannual steering committee meetings.

Custom Feature Users. On top of the basic controlling and management functionalities, the MaCoCo project offers additional features for specific use cases. One of those regards financial management between a faculty and its institutes, and another one regards additional features to handle large administration bodies (more than 500 users per institute). The wishes and requirements of these applications must be reconciled with the requirements of the mentioned groups.

The project aims to support the user in the day-to-day management of their institutes. Thus the platform had to be developed to incorporate their expertise and competence. An agile development process was used to include multiple user groups in the model-driven development process, to ensure development aligned with the end-users' needs.

The Software Engineering Process

To tackle the development of an application with such a large group of domain experts, we used an agile development method with an intensive and iterative requirements elicitation process that is based on the principles of Scrum [Sch97]. Figure 8.2 shows the iterative development process used in the MaCoCo project.

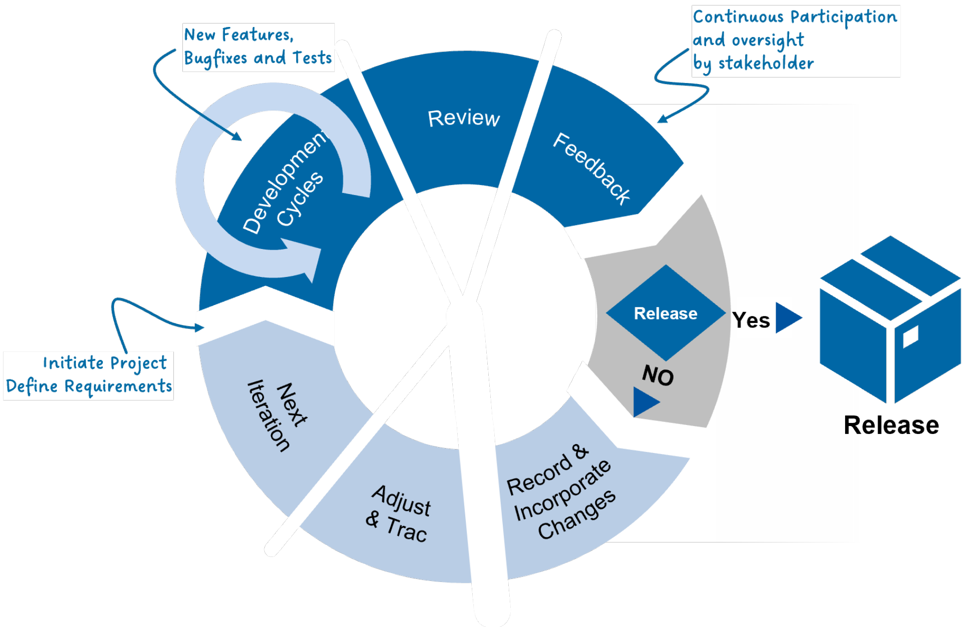


Figure 8.2: Agile software development method of the MaCoCo project [NGM+24]

An initial workshop was organized to gather, discuss, and prioritize all relevant requirements for the application. The input was clustered and features were defined. Three main aspects could be derived from the feedback: *Finance Management*, *Staff Management*, and *Project Management*. As a next step, concrete use cases, integrations, and corresponding tests are defined, and following scrum a backlog is set up with corresponding tickets. As a common practice in software development, these tickets are closed within development cycles. After the initial definition of requirements, a first prototype and next an initial data-structure model (root class diagram) could be defined, providing a basis for the generation of the application. The implemented features are then deployed on a test system where they can be reviewed by the corresponding stakeholders. Should the system meet all requirements defined in the current iteration a new version of the system is released, otherwise the feedback is aggregated and prioritized to be incorporated into the next build.
Figure 8.3 shows the growth of generated code within the MaCoCo project over time.

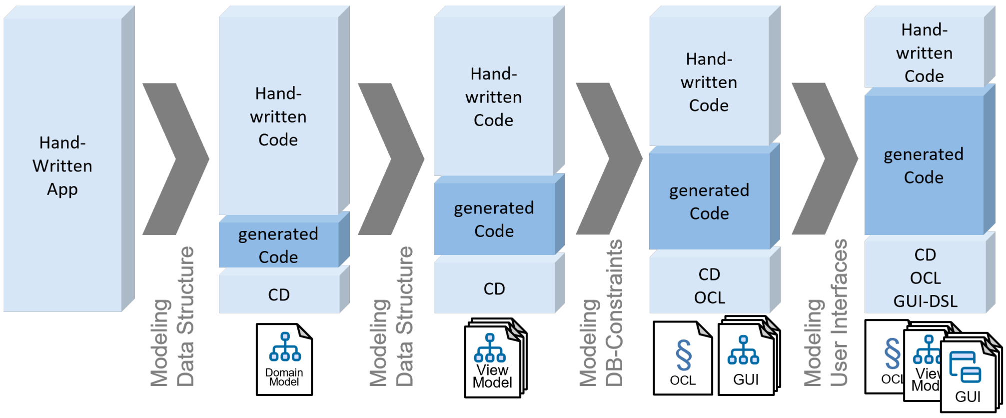


Figure 8.3: Retrofitting MaCoCo with generated code (adapted from [DGM+21]). Handwritten implementation is replaced stepwise with generated code and an increasing amount of models.

Starting from a prototypical handwritten web application architecture, a domain-specific model (CD) was derived. Using this model, basic code for data classes, persistence, and domain-specific core application code (*cf.* Section 7.3.1) was generated. Next, further systematic elements that were generated based on the domain model were identified. The generator was extended to implement the command pattern and generate data transfer between server and client (*cf.* Section 7.3.1). Support of OCL/P-Models was added next to generate data validation (*cf.* Section 7.3.4), and finally a specific DSL was developed: GUIDSL v1 in order to model user interfaces. At the time of writing, 3/4 of the code that makes up the web application is generated by the MontiGem-framework (*cf.* Table 8.1).

8.3 Requirements at MaCoCo

The primary features of the software encompass the support of faculties and chairs in their strategic and operational control and planning, more quality-assured processes, and improved coordination with the university administration [ANV+18].
The key problems MaCoCo tackles can be listed as

- Replacement of multiple individual management solutions that implement unknown quality control, with a standardized well-tested system: The majority of chairs use individual spreadsheet solutions that are prone to errors, tend to have no documentation, and have to be kept up to date individually.

- Professionalization of controlling processes, and provision of corresponding training and documentation.

- Improvement of process transparency while simultaneously improving security: A software solution can offer a fine-grained access concept, which in turn reduces the workload of an administrative office by giving individuals the opportunity to manage their own data (e.g. self-service).
- Standardization of overall processes and thus eased communication between administrative entities.

MaCoCo aims to provide institutes with the means to overview, manage and plan with its resources.

8.3.1 Technical Requirements

MaCoCo is a platform that is being used on a day-to-day basis outside of lab conditions in the 'real-wold'. Thus additional requirements can be derived that are independent of the specific use cases of the project, but can be linked to the targeted high technology readiness level TRL 9:

Req.M1 **Data Privacy:** As the platform handles very sensitive data, a system has to be established that allows the restriction of data to only be visible to authorized personnel. It should be transparent what access the user has to what data at any point in time.

Req.M2 **Uptime:** The platform is provided to the user at any time. Thus monitoring has to be set up to enable faster reaction time to any server issues. A method has to be established to inform the users of any upcoming downtime of the system.

Req.M3 **User Support:** The platform has to provide means to support the user while using the system. This could be provided by direct communication channels (e.g., email, chat, telephone) or by providing documentation.

Req.M4 **Usability:**The platform is intended to replace existing solutions, therefore it has to be implemented with usability in mind. If the platform delivers the required functions, but not with sufficient usability, it will not be accepted by the users.

Each chair needs its own setup in this system and might introduce new requirements at their setup over time due to different outside factors. Agile software development helps to quickly meet these changing needs and the model-driven methodology used in this project supports their timely realization.

8.4 Implementing a Model-Driven Real-World Application

MaCoCo is realized using the MontiGem-framework (*cf.* Chapter 7). Thus the application can be represented with a three-tier architecture [Eck95] consisting of a *presenta-*

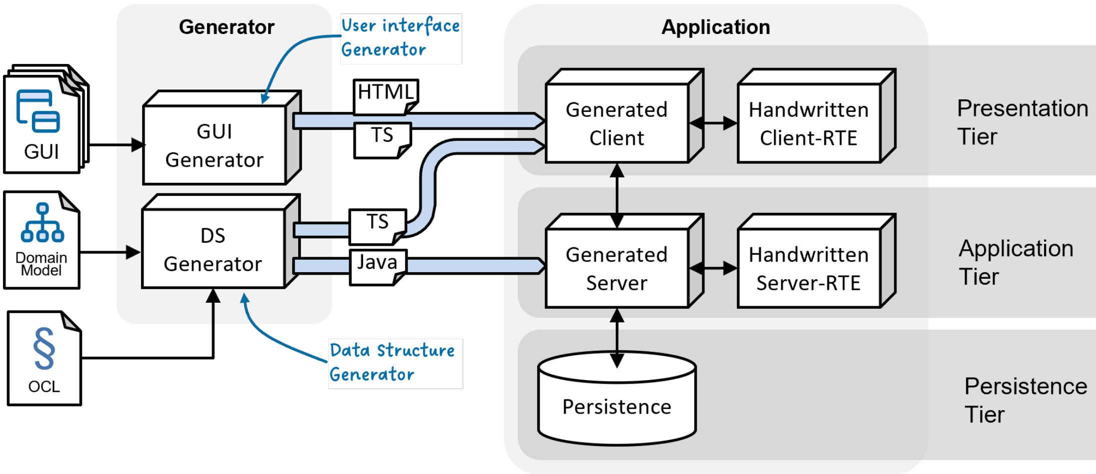


Figure 8.4: Model driven development of MaCoCo using multiple generators to create a web application.

tion layer (client) an *application layer* (server) and a *persistence layer* (database). The project was started by implementing a handwritten prototype, outlining the key concepts an scope of the system. Based on the feedback of developers, lead users, and the steering committee of the project, a generator was developed and used to replace and extend existing code step by step. Although the development of a generator seems to be unnecessary extra work, as it only replaces existing code, it brings great benefits in the later development phases of the project, as code can be modified and extended much faster [AMN+20]. In order to keep the data modular and to increase security, each chair has its own database within MaCoCo. Following the MontiGem-approach (Figure 8.4), a set of models is used to generate the majority of the code [ANV+18].MaCoCo uses CD4A class diagrams (*cf.* Section 3.4), GUI-models (*cf.* Section 5.1) and OCL models to define the target application.

The controlling management of a chair can be divided into three major aspects: *financial management*, *staff management* and *project management*. The different aspects are highly interdependent and interwoven with each other. Within MaCoCo, these aspects can be seen as different views upon the same management challenge: From a financial management point of view a bank account has bookings for salaries from multiple employees that are booked every month, from the view of staff management, the salary of an employee is financed via multiple specific accounts. From a project management view, a third-party project allocates funds within one bank account that is used to finance an employee who is assigned to that project. One of the great benefits MaCoCo provides, is the system-wide data consistency and transparency over the financial processes. The user has one tool, that allows him to observe, plan and manage the relevant controlling

aspects of the chair.
Over the years MaCoCo has gained a lot of complexity. The application comprises more than 60 pages (Figure 8.5) that manage over 100 persisted classes for multiple databases.

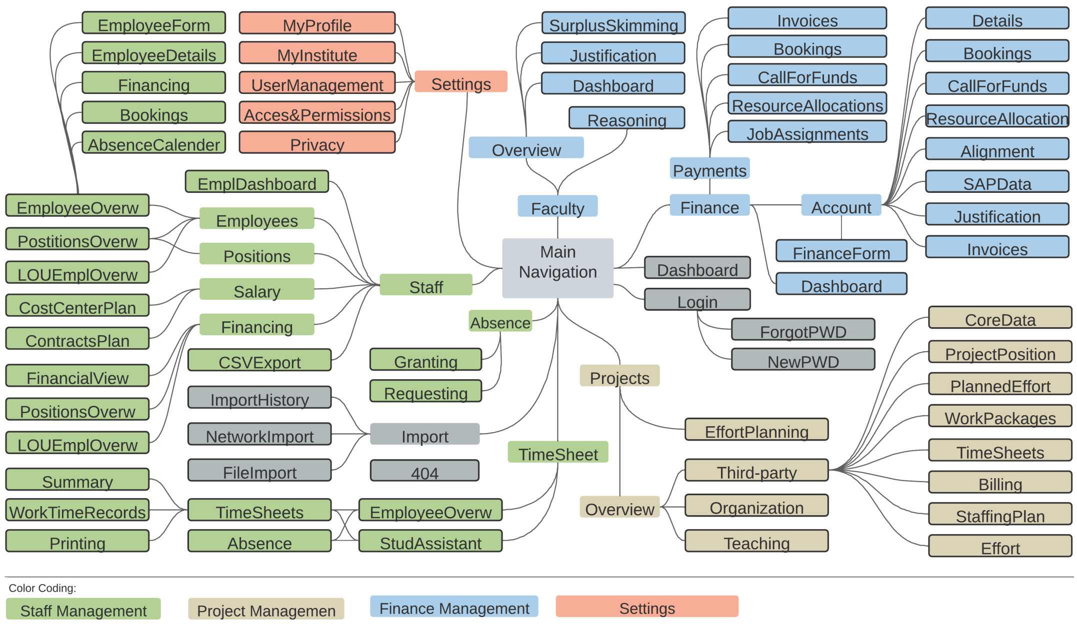


Figure 8.5: Overview of generated pages in MaCoCo and corresponding navigation between them (July 2022).

The layout and content of the pages, with the exception of the navigation and header bar, are defined by GUI-models. Figure 8.6 shows a screenshot from MaCoCo. MaCoCo presents the same look and feel as MontiGem due to the similar architecture. However, through daily use and iterative lead user-driven optimization, the models and components are more refined and show in general a higher complexity. The MaCoCo application tends to follow Schneiderman Mantra [Shn96, CCOTF09], implementing (at least in part) the seven tasks proposed by Schneiderman (*cf.* Section 6.2), which we discuss in the following:

Schneidermans Mantra: Overview

One of the principal goals of MaCoCo is to provide the user with an overview of his managed data. This challenge is met by providing both dashboards and overview pages (Figure 8.6). The dashboards are used to provide relevant data about different data types at a glance, whereas overview pages provide insights into the data for one specific data type.

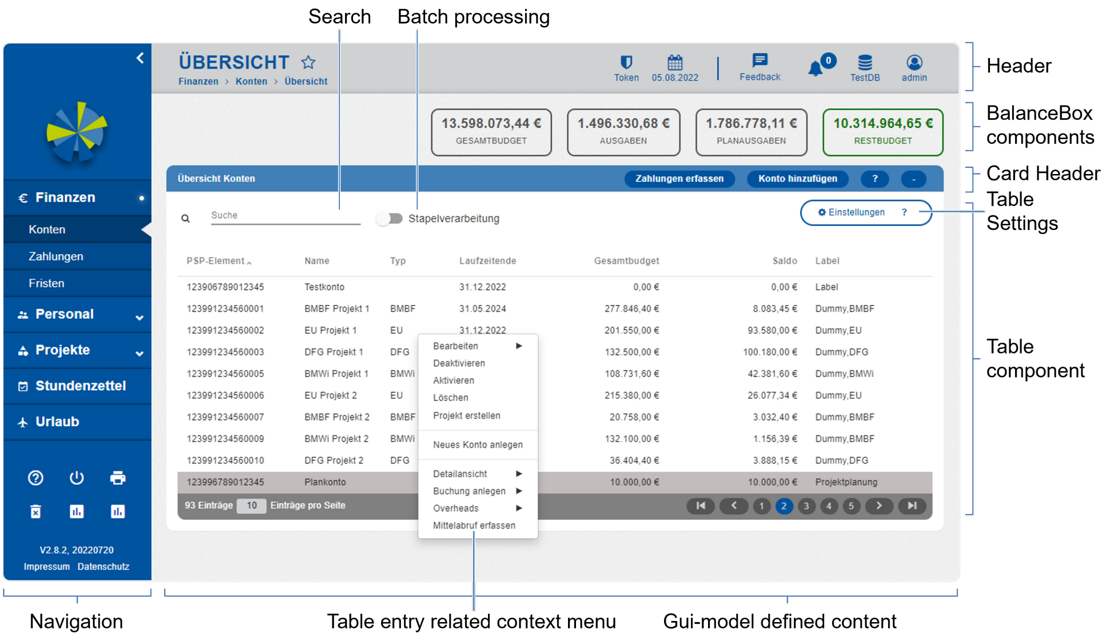


Figure 8.6: GUIDSL v1 based Account Overview Page. The page is defined using a fully configured table component (*cf.* Section 5.1.3) and following the structure of the Overview Page as provided by CD2GUI (*cf.* Section 6.2.2). The corresponding model is shown in Listing A.5

Schneidermans Mantra: Zoom

MaCoCo supports zooming only in part. 'Graphical' zooming as described in [Shn96] is supported by default by any current browser, and will only yield a limited gain in information, as the user can adjust the information density on his screen by zooming out, but will not see any new information by zooming in. However, MaCoCo also allows the customization of the fields shown within a table, thus tables can initially show a reduced overview, if needed the user can display further information, hence 'zooming in' on specific data points.

Schneidermans Mantra: Filter

In order to permit quick identification of relevant data, tables in MaCoCo support a simple filtering search mechanism. It allows to search all data within the table and thus filters out any non-matching results. On several pages 'advanced' searches were implemented that allow filtering not only for displayed information but also for specific context of the displayed data (8.8(b)). E.g. filtering a list of time sheets by the related project.

Schneidermans Mantra: Details-On-Demand

The platform provides multiple pages that provide detailed information and context of specific objects (*cf.* Figure 8.7, Figure C.3). As MaCoCo focuses on the management of finance, staff and projects. There are individual pages to manage accounts and budgets, staff and their contracts, as well as projects and related resources. Each details page provides the user with options to inspect the object in focus and if needed change its parameters (e.g. change the account through which a project is billed, or add bookings to an account).

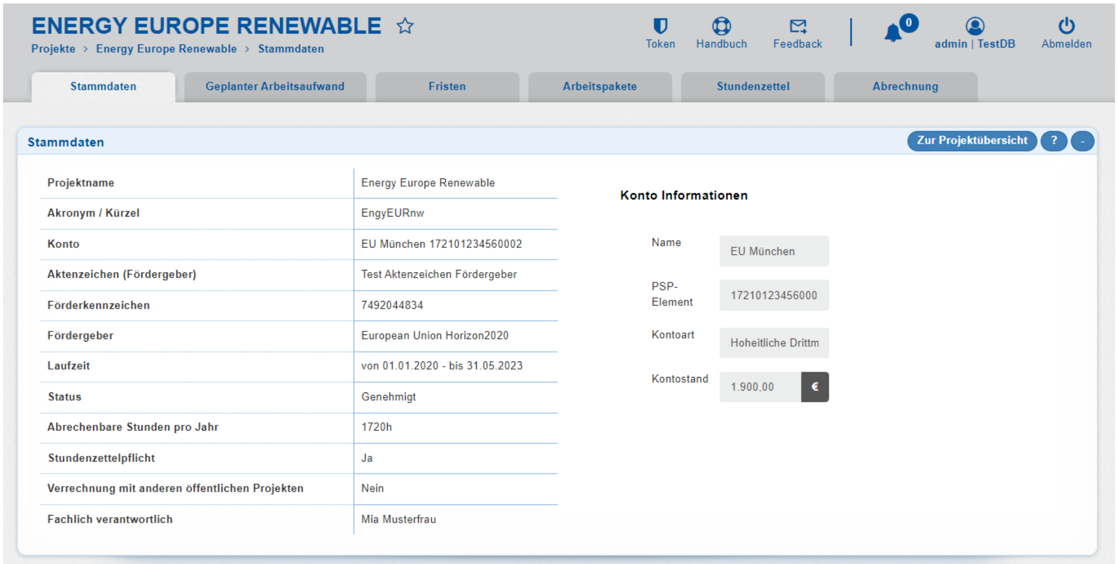
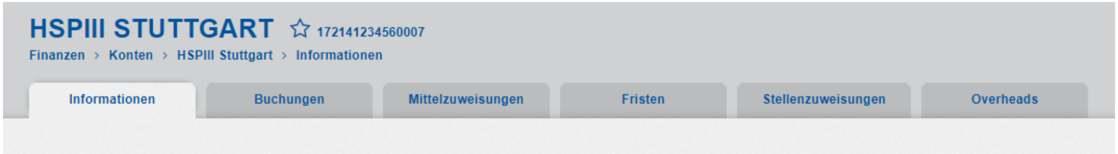


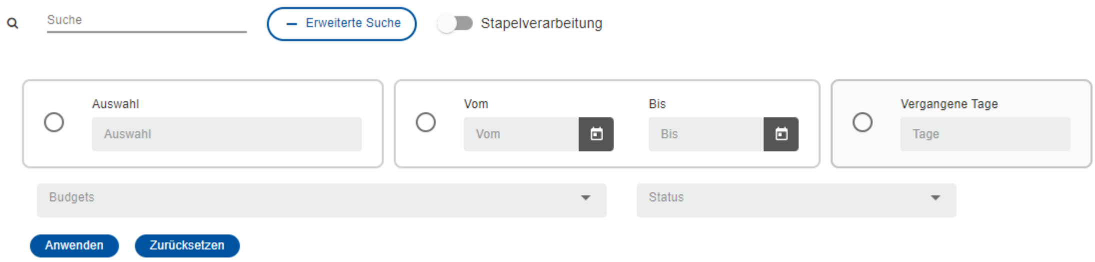
Figure 8.7: Screenshot of the *details page* of a project in MaCoCo, giving an overview of the attributes and linked employees as well as the option to modify the project.

Schneidermans Mantra: Relate

Relations between objects are rarely visualized within the overview pages, as the usable screen space is limited. In MaCoCo relations of one specific object are shown within their configuration and details pages. Depending on the complexity of the related objects further pages were added. The bank account is strongly related to bookings, invoices, and further financial flows, thus there is a separate page to inspect these specific relations (*cf.* Figure 8.8(a)). The platform provides multiple pages that provide detailed information and context of specific objects (*cf.* Figure 8.7).



(a) Further relations to other objects are visualized in separate pages



(b) Extended Filter options, allowing for more advanced searches

Figure 8.8: MaCoCo screenshots (MaCoCo Version 2.13.3, December 2023)

Schneidermans Mantra: History

MaCoCo provides an event history listing changes on the database. However, due to the complexity of undo mechanisms in a multi-user web application and the lack of user requests for this feature. It is not yet implemented. The current history feature lets the user observe precise changes to any data to which he has access.

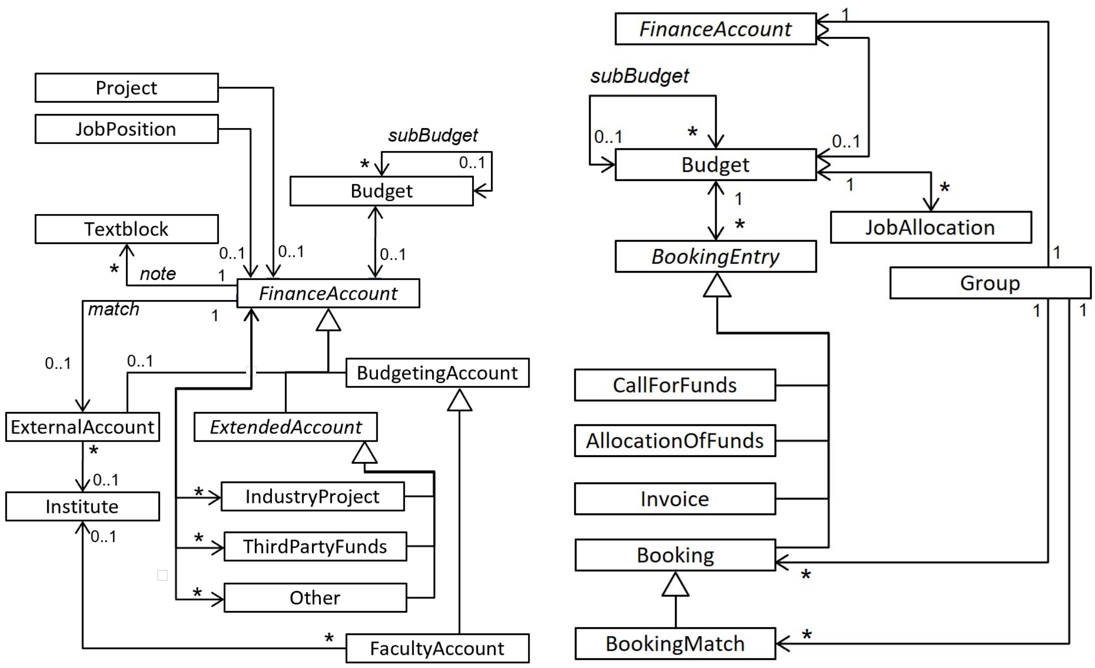
Schneidermans Mantra: Extract

MaCoCo has several data exports implemented. As a general feature, any data table can provide its data as a CSV file to the user if not otherwise prohibited. Cases that are prohibited include sensitive staff data that is regulated by data privacy policies. Additionally to this export, MaCoCo provides several APIs that provide reports on specific data sets, as well as PDF exports for time sheets and other printable tables.
As MaCoCo implements the same architecture as MontiGem, it has a very similar look and feel. Due to its optimization for daily use, there are some additional components and extensions, compared to MontiGem: Tables in MaCoCo provide a toggle switch to enable batch processing of table entries. MaCoCo-specific components such as 'BalanceBox' where added. This specific component is optimized to display one specific value to the user and is used throughout the platform to differentiate between key values and less relevant data. MaCoCo also makes use of the context menu in many places to give the user alternatives to perform his tasks. The context menu is a 'hidden' menu, but has the great benefit of providing functionality related to the 'clicked' element. This less

intuitive menu was added to MaCoCo to support faster GUI interaction for daily users. In the following we will take a closer look at the implementation of the three different aspects of controlling management of a chair: *financial management*, *staff management* and *project management*. We also take a closer look at the specific model used to define the application. The complete models can be found at: Listing A.1 and was published in [GHL+22].

8.4.1 Modeling Financial Management within MaCoCo

As an enterprise resource planner one of the key feature of the platform is financial management. MaCoCo provides overviews and insights into the current financial situation of the chair. This includes current bank accounts and planned accounts, predefined and custom budgets and sub budgets as well as multiple types of billed and planned bookings and invoices. The financial situation of a chair is determined by many factors and external influences. MaCoCo must be able to process the majority of these in order to enable the user to accurately plan and perform informed decisions on the finances of the chair.
The finance section provides MaCoCo users with planning, management and control of financial resources. By aligning the account structure with the project types, it is possible for chairs and institutes to manage the financial facts of the projects in individual accounts. Here, the expenditures and revenues within the scope of a project can be inspected and configured. Thereby the expenditures of a project are represented via bookings and the revenues via calls for funds, invoices and allocations. The expenses of a project can be divided into cost categories in MaCoCo by creating budgets and distributed to individual years. This gives the chairs and institutes an extended overview of the expenditures and allows for more detailed planning. In addition, chairs and institutes can see at any time which funds have been spent, which funds are planned to be spent, and which funds are still available. The combination with the staff area enables the planning of employee expenditures and funding. The employer debits of the employees are automatically posted to the respective accounts by the staff administration in the staff area. With the help of an interface to SAP, the actual values are imported into the respective accounts. This allows the users to control the expenses accordingly and to compare them with their planning. Processes such as overhead management are also automated in MaCoCo. The stored information in the accounts automates the budget-effective returns of the overheads when creating new revenues. The rights and roles concept further enables information to be made available to specific user groups in a targeted manner. This makes information easily and quickly accessible at all levels in the departments and institutes.



(a) Account-Classes

(b) Booking-Classes

Figure 8.9: Excerpt of the MaCoCo data structure model. The diagram above omits all attributes in order to increase readability. The complete CD is presented at Listing A.1 and published at [GHL+22]

Single Underlying Model for Finance

Within the architecture of MaCoCo's financial data model, the `FinanceAccount` class, depicted in Figure 8.9(a), holds a central role. Each `FinanceAccount` may be associated with a `Budget`, which itself can encompass several sub-budgets. Additionally, budgets are capable of containing numerous Bookings, a relationship showcased in Figure 8.9(b). A noteworthy early development feature was the incorporation of `Notes` to be attached to accounts.

To cater to a diverse array of financial use cases within MaCoCo, the `FinanceAccount` has been extended into several specialized classes. It has been refined into three distinct subclasses under the `ExtendedAccount`, which offers more detailed account information. The classes `IndustyProject`, `ThirdPartyFunds`, and `Other` are designed to serve the primary financial operations of academic chairs and institutes. Additionally, the data model accommodates classes for synchronizing with the SAP financial system and for managing communication between the faculties and institutes. Accounts sourced from SAP are

handled within the unique subclass named `ExternalAccount`.
As the project evolved, the `FacultyAccount` was added, designed to adhere to the internal policy of a faculty that limits the proportion of the annual budget chairs can retain at the year's end. The dean's office oversees these accounts to ensure compliance, engaging with institutes to either spend the remaining funds or provide a rationale for any surplus. The `FacultyAccount` thus includes elements for managing such communication, and is linked to specific institutes to give context for the faculty members, reflecting the multi-institute interactions of a single faculty.
MaCoCo's transaction tracking utilizes the `Booking` objects, with the structure for this class illustrated in Figure 8.9(b). Originating from the abstract `BookingEntry` class, the `Booking` class is designed to be inclusive of each financial entry, which is invariably associated with a budget or sub-budget. To meet the varying needs of the application, the `BookingEntry` class has been extended variously. To align local accounts with SAP at the booking level, additional information is stored within the `Booking` class, and external data alignments are tracked using the `Group` class.
Moreover, the financial data framework of MaCoCo is seamlessly integrated with other sub-domains. Accounts are directly linked to projects (see Section 8.4.3) to streamline the referencing of financial information within project contexts. This association is unidirectional to optimize data retrieval efficiency, as accounts typically hold relevance within the scope of projects. This integration extends to staff management (detailed in Section 8.4.2), where accounts serve as financing sources for `JobPositions`, and budget allocations for `JobAllocations` are specified to manage staff-related financial transactions.

Graphical User Interfaces

MaCoCo has 19 pages that focus solely on financial management (Figure 8.5). With the exception of one page all are modeled with the GUIDSL (Section 5.1). In the following, we will take a look at a few of the modeled pages.
Finance Overview Dashboard

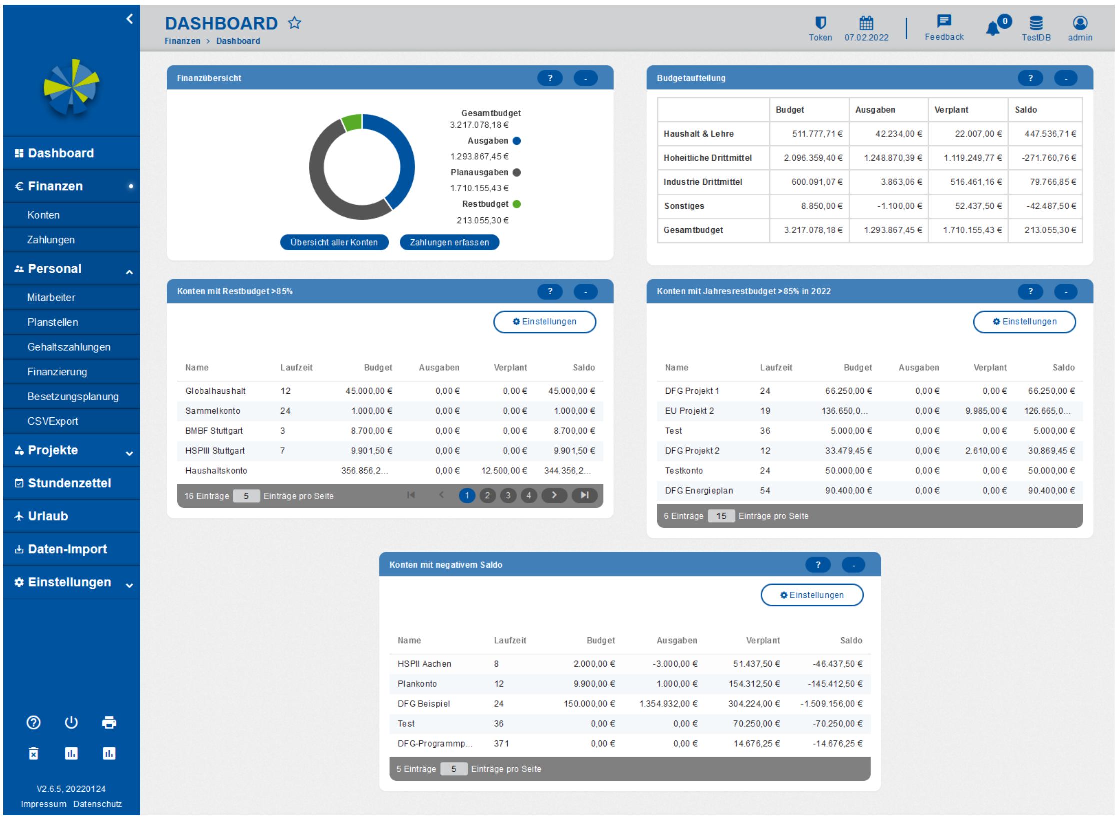
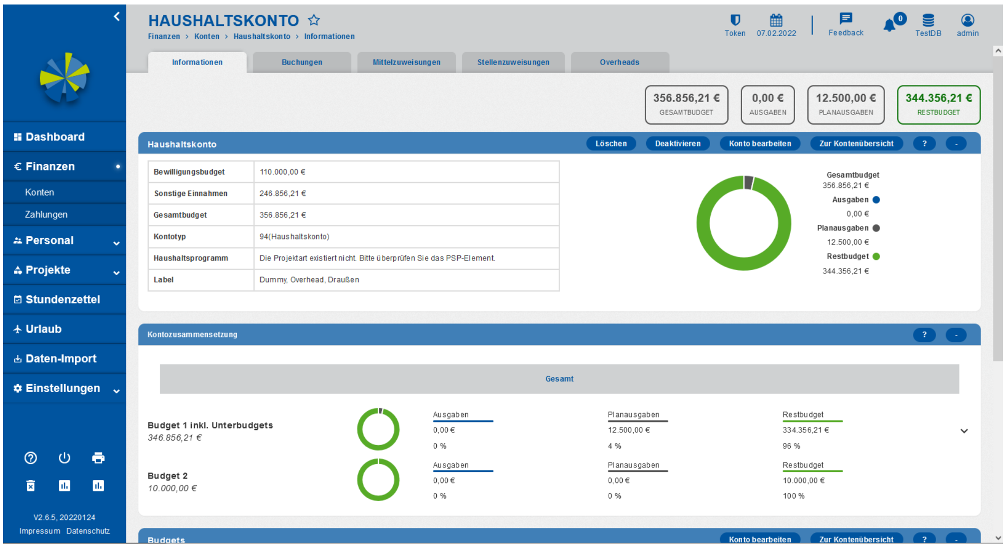


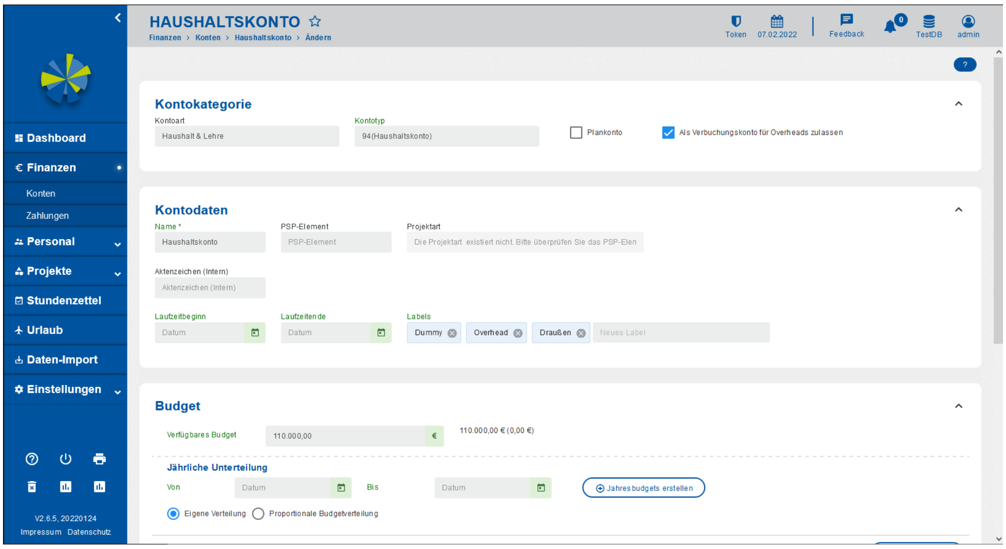
Figure 8.10: Dashboard showing relevant information about the financial status of the institute. Top left: Pie-Chart displaying the aggregated budget. Top right: Aggregated finances for different types of accounts. Bottom left: Account with more than 85% budget remaining, Bottom right: Accounts with more than 85% of annual budget remaining. Bottom center (partially occluded in screenshot) Accounts with negative balance.

The finance dashboard (Figure 8.10) provides the user with a very specific overview to manage the finances of the chair. Based on user feedback, the dashboard has been optimized to show mainly accounts that are relevant to the currently logged in user. Thus, there is no list showing all accounts on this page. As the dashboard is intended to be used as a starting point, it provides only navigation options to the displayed elements. This option was chosen to keep loading short and keep from developing multiple editing pages in parallel.

Account specific pages



(a) Detailed view for one account, showing balances, budgets and editing options. Tabs in the top row show various views for the account such as bookings ('Buchungen') or cash flows ('Mittelzuweisungen').



(b) Edit-Form, to create or modify accounts. Here keydata ('Kontodaten') can be set, such as 'Name', 'PSP-Element' or running time 'Laufzeit'. Furthermore budgets and subbudgets are defined here.

Figure 8.11: Two of the account specific pages for financial management. Further pages are: 'Bookings', 'Job assingments', 'Overheads', 'Resource Allocation' and 'Invoices'

Each account has its own details page (*cf.* Details Page: Section 6.2.3) in MaCoCo (Figure 8.11(a)). It shows the budgeting over multiple years, the balance type and cash flows of the account, and offers the possibility to record short notes for this element. Besides the account detail page, there are many other pages to manage the integration of the account into the chair processes, such as job assignments and third-party funding. A very important page is the edit form for the account (Figure8.11(b)). it remains one of the more challenging user interfaces of the MaCoCo project and is one of the few user interfaces that isn't generated. The form enables the user to create or edit accounts, and all it's related complex attributes. Since the account is interleaved with many other objects, the account form contains a lot of logic to ensure that the object to be saved is valid. In addition, the user is supported early on in his creation process, so that he does not spend a lot of time in the configuration of an unsavable object. The user is informed as early as possible about possible inconsistencies.

Account independent pages A user request early on was the option to manage finances independently form a specific account. Pages were needed, that list all bookings (Figure 8.12) invoices, resource allocations, calls for funds, and job assignments (*cf.* Overview Page: Section 6.2.2). Due to the high quantity of data and a focus on displaying relevant data as fast as possible, an adjustable filter on the last 30 days was implemented. While the previous pages focused primarily on individual adjustments, these pages support the user in the processing of large amounts of data. For this purpose, batch processing was integrated into the tables. It allows the user to select any number of elements and modify them in the same way. For example, to move 25 entries from one budget in one account to another. The user is provided with a tool, that enables him to potentially load all bookings from the database. As this can take a highly uncomfortable loading time, a filter option is mandatory to ensure usability. However, the user is overwhelmed when being prompted with a highly configurable filter, which results from all feature requests from different use cases. The solution was found in a well-defined default filter: The user is provided with bookings of the last 30 days. This captures the majority of use cases and excludes only some rare cases where the user has to look at bookings from the more distant past. One of the major challenges of the MaCoCo project is to bring together a large number of very different use cases under a single platform, which is expected to be intuitive and highly efficient at the same time.

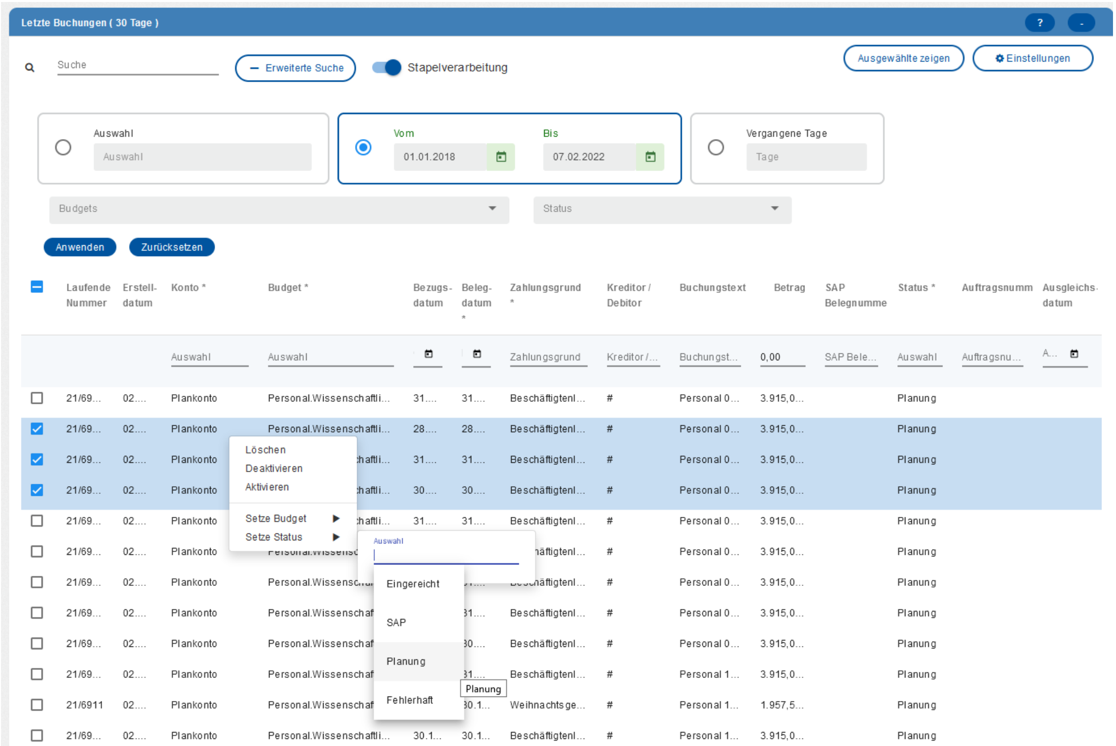


Figure 8.12: (MaCoCo Screenshot) Page showing the adapted filtering (Collapsible element in the header of the table) and batch processing of multiple bookings at the same time. Multiple lines can be marked and edited via the context menu.

8.4.2 Modeling Staff and Human Resources within MaCoCo

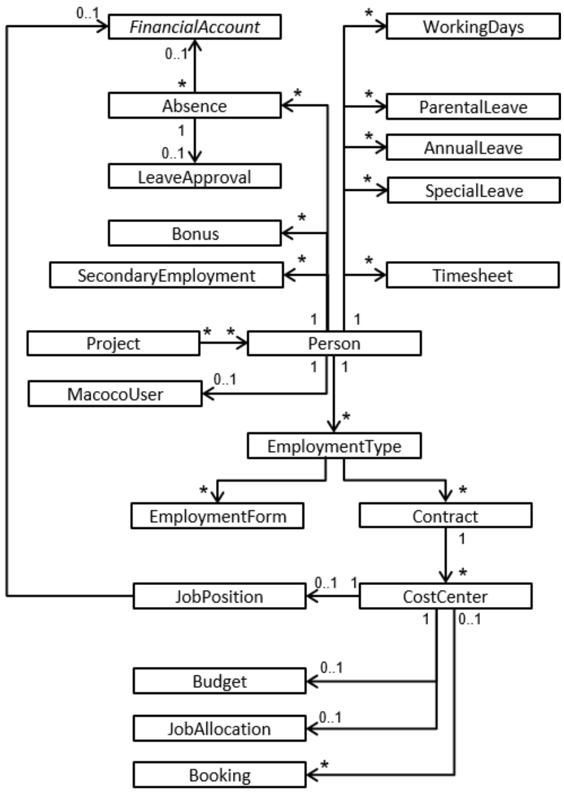


Figure 8.13: Class Diagram for Staff management within MaCoCo. The complete CD is presented at Listing A.1 and published at [GHL+22]

With 607.0 million € volume of third-party funds[5] making up 51% of research funding in 2022 more and more staff in universities are funded externally, therefore leading to the requirement of more pronounced planning, monitoring, and control of staff financing. As of February 2024 the budgets managed by MaCoCo amount to a total sum of more than 2 Billion[6] Euro. In the 'Staff management and Human Resources' area, management, planning, and controlling of employees of a chair or institute is originated. Here, employees can be created and edited. The creation of an employee takes into account the specifications of an employment relationship at a university. Thus, different occupational groups can be created, in which employment at a university can take place. It is also possible to map the general conditions in terms of wages and compensation. Since

[5]https://www.rwth-aachen.de/cms/root/Die-RWTH/Profil/ enw/Daten-Fakten/?lidx=1
[6]Sum of all overall budgets without test data as of 11.2.2024: 2.188.286.257,17 EUR

employees at public universities in the state of North Rhine-Westphalia are employed in the public sector, the relevant collective bargaining agreement applies to these employees. This is based on a remuneration table for determining wages. The remuneration table consists of various remuneration groups and levels in which an employee is classified depending on the type of employment. This classification can be made for each employee individually. The contract management of an employee is done by creating contracts within the employee editing form. The financing of the employee can then be stored in this contract by means of cost centers and various cost center types. Similarly, personnel postings are created, taking into account the posting period and the posting scope. The personnel postings are then booked directly to the respective stored account and budget. This facilitates the planning of employee financing. Tables of employee salary payments and financing also serve to support decision-making and transparency through the management cockpit functionalities.
It is also possible to store absences for employees, e.g. vacations. This is accompanied by an overview of the available vacation days and the vacation days taken. These absences are transferred to the timesheet area for plausibility checks so that no working times can be recorded on them. This reduces the risk of submitting time sheets to funding agencies that are not true. This is to avoid allegations of grant fraud.
MaCoCo provides several tools for personnel management. As MaCoCo is also a planning tool, different financing options for employees can be determined here. MaCoCo supports the user and points out problems in financing. The management of employees presents new challenges for the developers of the platform. In addition to the typical efficiency requirements, there are data protection regulations that must be strictly adhered to. Staff management is strongly linked with finances due to the salaries of the employees Figure 8.13. We will look at a few of the implemented use cases below.
Employee Overview and Position Overview The MaCoCo platform distinguishes between employees and the positions in which they are employed. This enables project-specific planning of funding for specific employees. Through this, especially errors in planning can be detected early, both in case of over- and underfunding.
Funding An important aspect of financial planning is the coverage of expenses. The segment funding aims to provide tooling to optimize the cash flows in order to have all staff expenses covered. Here, different views on the finances are offered: Which accounts and how they are debited, how employees are financed, and whether positions have financing (*cf.* Figure 8.14).
Employee Details The overview pages list staff and provide an easy link to the details pages for one specific employee. There are four views for each employee. The first is the detailed employee view, showing key parameters such as the time interval of employment, kind of employment, and accounts the employee is financed over. The second view is employee funding, as shown in Figure 8.15(a), which includes cash flows and coverage of salaries. The third aspect is bookings, which lists all bookings related to the salaries of the specific employee. Lastly, the absence calendar shows the working time distribution

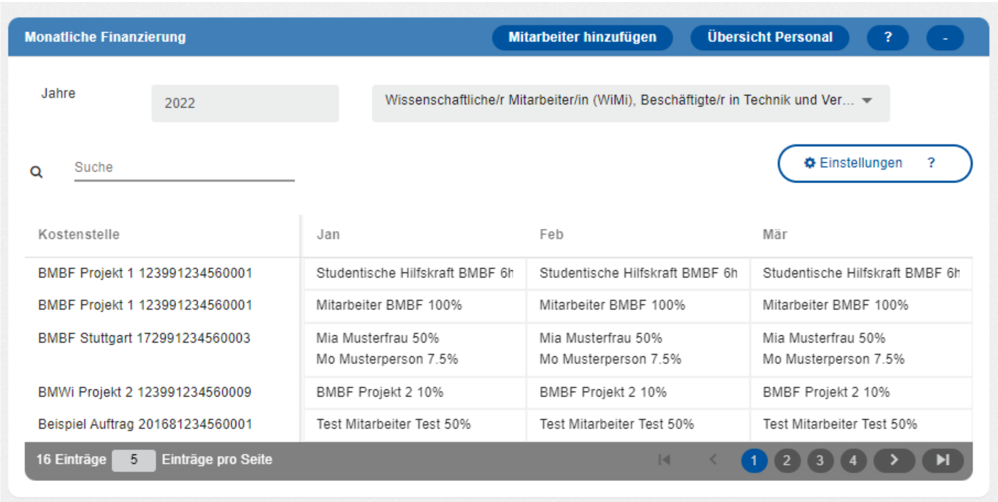


Figure 8.14: Table in MaCoCo showing a mapping between users and projects for each month and their respective coverage of funding.

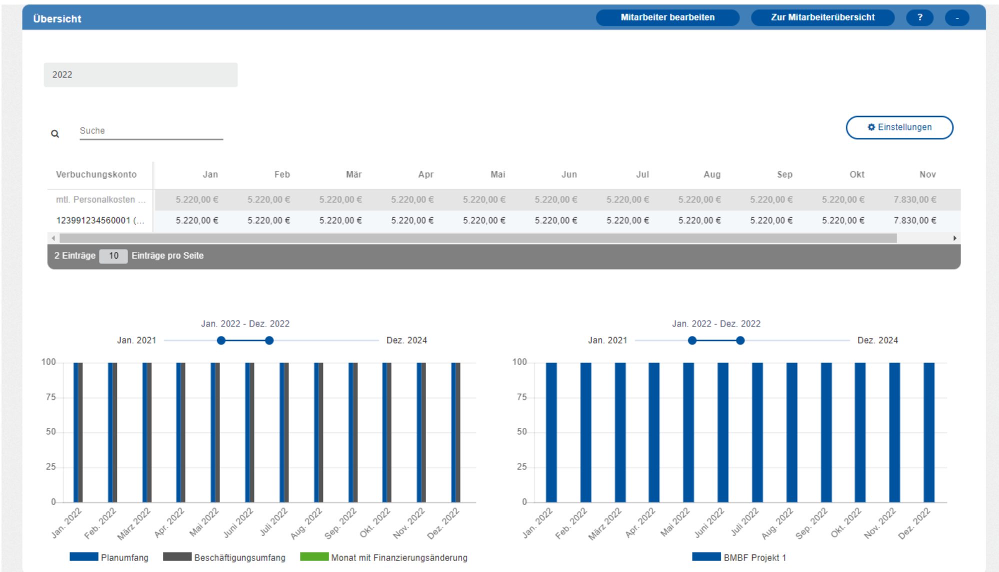
and current requests for leave.

Salaries This segment gives the user an overview of the actual expenses for the personnel over a certain period of time, i.e. the bookings for the salary and the planned expenses for the personnel determined by contracts.

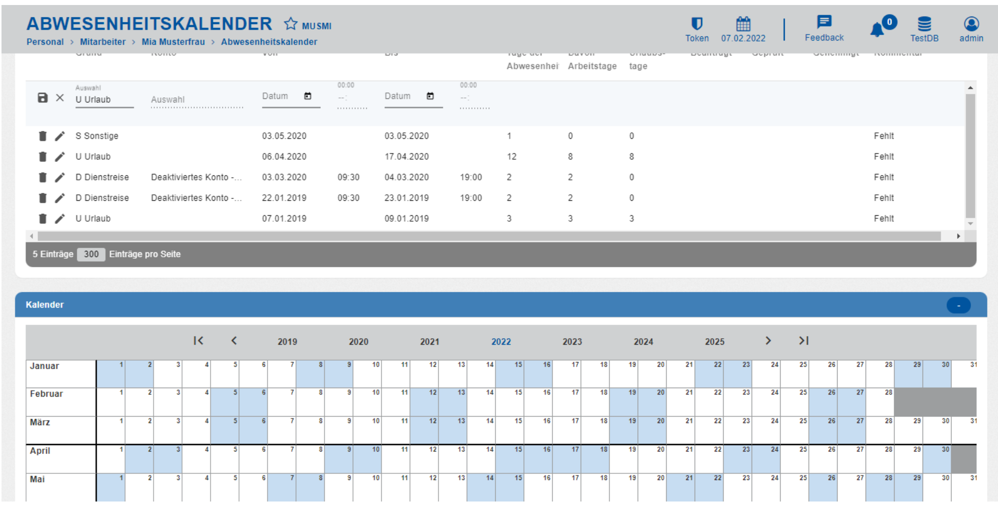
Staffing Plan In academia based on research projects and contract negotiations, there is a limited amount of fixed positions that can be filled with employees. Well-planned staffing can save the institute a lot of money here, or in contrast, the institute might give away a lot of money if does not plan carefully. It is provided with the financing for these positions whether they are filled or not, thus these important views were added in order to help the user to keep these positions staffed at all times.

8.4.3 Modeling Projects and Time Tracking within MaCoCo

A lot of research by institutes can be financed over third-party-funded projects. These come with their own requirements, such as custom time sheets. MaCoCo offers tooling to support the user in keeping track of currently running projects, which employees are currently working on them, and what resources are planned to be spent on it. Figure 8.5 shows how the pages in MaCoCo are set up to provide an overview based on the use case the manager has at hand. If the intention is to manage the workload of the employees, projects can be viewed based on the relation between employees and the expected workload for currently running projects. If the manager needs to organize the projects of the institute, he can also filter by the type of the project and see if there are still third-party projects that are still pending.



(a) (MaCoCo Screenshot) Funding of one employee. The (exemplary) employee is financed to 100% within the shown time interval from Jan 2022 to Dez 2022 over the project 'BMBF Projekt 1'. The exact bookings and corresponding accounts can be seen in the top table.



(b) (MaCoCo Screenshot) Calender of absences. Shows the planned leave, and remaining vacation days for one employee.

Figure 8.15: Two of the employee-specific pages for staff management. Further pages are: 'Overview' and 'Bookings'

All these overviews lead to specific projects that can be inspected and edited in detail over a set of views. The time sheets of the employees are kept in the time sheet area of MaCoCo (Figure 8.16. Here, the working hours can be recorded for a project and the respective work packages. For this purpose, target hours are calculated for the employees on the basis of the contracts. With the help of the recorded actual hours, overtime can be calculated. The time sheet section complies with the requirements of the Minimum Wage Act. Certain employees are subject to time sheet requirements and must record their working hours. These working hours must be kept by the employer. This is taken into account by means of centralized time sheet recording, which can be carried out by each employee. Furthermore, the time sheets are used to prove the working hours to the funding bodies. For this purpose, time sheet templates are stored for individual funding bodies, which take into account the requirements of the funding bodies. For this reason, the plausibility check is of great importance. The absence times are therefore automatically transferred from the personnel area to the time sheet. Since in the time sheet area the hours can be recorded on projects and work packages, the number of recorded hours on the individual work packages is possible for a project. This makes it possible to compare the planned effort with the actual effort incurred and to control it in the event of deviations.

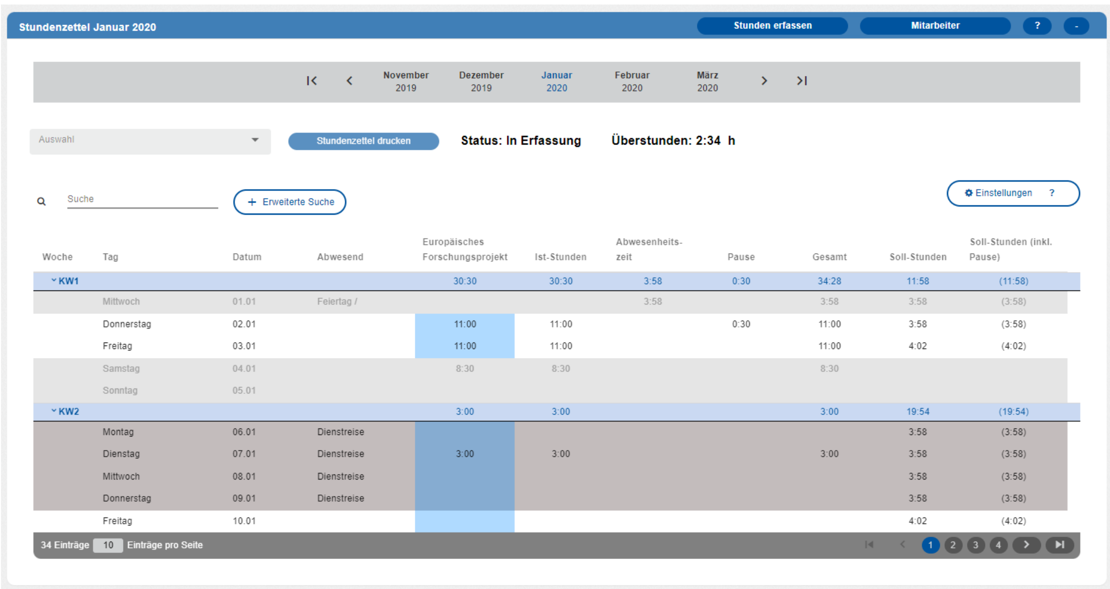


Figure 8.16: Screenshot of a Time sheet as implemented in MaCoCo. The time sheet indicates, what time was spent on which project. It indicates the work time, break time, and required time. In addition, time absent (e.g. business trips) and holidays are visualized. Although being a complex table, it still provides all default component features such as search and filtering.

8.5 Extending MontiGem to Generate MaCoCo

MaCoCo uses MontiGem to generate the web application. Throughout its development MontiGem was iteratively reconfigured, extended, and modified in order to comply with the demanding requirements of the application. Many of the changes were incorporated into the development of the generator and now form part of the core of MontiGem. In the following we will take a closer look at the changes that are primarily focused on the MaCoCo use case.

8.5.1 Used Domain-Specific languages

As MaCoCo and MontiGem share a common architecture, the DSL CD4A (see. Section 3.4) is used similarly. In MaCoCo there are three types of class diagrams: (1) The Domain model introduced in Section 7.3.1, (2) The View models, as described in Section 7.3.2 and (3) the command models (see. Section 7.3.3). Similar to MontiGem the class diagrams are used to generate domain-dependent artifacts of the persistence layer, classes to access and move data from the server to the client, and multiple helper classes, such as validators, commands, and builders. In order to meet the requirements of the different use cases and the increased need for performance of MaCoCo, the generator in MaCoCo has been expanded multiple times compared to MontiGem.

Generating Additional Code for Data classes:

Labels Very early on in the project, lead users expressed the wish to have the option of labeling and managing different objects themselves. As this applies to almost any object the user interacts with the generator was extended to add a `List<String> labels` to all data classes. Allowing the developer to persist any comments or labels to any modeled class.

Diff MaCoCo is a platform on which many users cooperatively manage their data. A log that is understandable to the end user is a very useful tool that provides transparency since administrative processes can be quite complex and often span multiple user roles. Both MaCoCo and MontiGem use classical logging to track errors and debug information, but this information is too technical to be provided to the end user and might contain sensitive information. Thus an end user-friendly log is needed, that only provides information that is relevant and within the restrictions of the user. In order to provide this log, each class was provided with methods to track any changes to an instance of that class: `List<String> getFullDiff(Object o)`.

End User-Friendly Designations Another extension to the generator of the domain class, was the addition of end user-friendly designations for attributes and classes. Many classes have technical names that are not meaningful for the end user or do not match the naming conventions the end user is used to in his day-to-day work. In order to manage this discrepancy, additional customizable designations were added via the generator that can be used in any instance the system provides the user with information regarding a specific class: eg. Error messages or notifications: "Could not delete 'Bank Account'".

Generating Additional code for Data Access Objects One of the more significant differences between MaCoCo and MontiGem are the performance requirements, especially when processing large amounts of data. Tests and run-time analyses have shown us that it is primarily the interface to the database (JPA and Hibernate) and the generic accessing of the database that slows down processing. For this reason, the generator was provided with additional methods to enable access to the database using methods adapted to the current use case.

Lazy Loading There are two prominent loading strategies in web development: lazy loading and eager loading. *Eager loading* can have a negative impact on performance if applied in large data structures. Depending on the data structure significant portions of the database might be loaded in order to retrieve all objects that are linked directly or indirectly to the eager-loaded object. In contrast *Lazy Loading* only loads data that is relevant to the calculations at hand, thus taking less time and having less impact on performance. MontiGem provides both approaches in its data access objects (see 7.3.1), however, there is an option to specify the depth to which an instance of a class and its linked objects are loaded. Thus either only the instance itself is loaded or all associated objects are loaded as well. These options suffice for small-sized applications as they do not have too large databases and data structures. Large applications however will suffer from a high impact on performance each time the developer is forced to use the eager loading approach. To mitigate this problem an option was added to define the depth to which linked objects will be loaded. This enabled the developer to load only the required objects needed for his use case without loading unnecessary objects. Figure 8.17 shows the data structure of an bank account object in MaCoCo. The numbers in the dashed boxes indicate the loading depth that is required to load the linked object. Lazy loading up to a depth of 1 is equivalent to lazy loading in MontiGem, lazy loading up to a depth of 2 would return the bank account object together with the linked objects: comment, TotalBudget and ExternalAccount. Lazy loading up to a depth of 3 is equivalent to eager loading in MontiGem (in this example),

OCL in MaCoCo

Constraints upon the data structure of MaCoCo can be defined via OCL. The data structure generator matches the classes defined in both OCL models and class diagrams and creates validators for both the back-end and front-end.

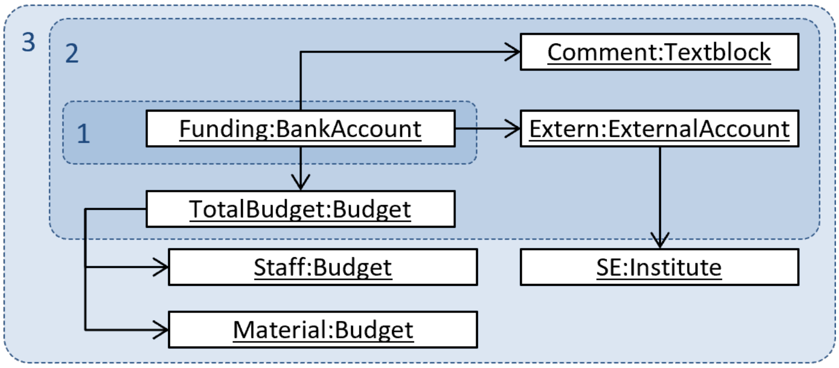


Figure 8.17: Example for loading depth applicable in generated lazy loading strategy

OCL-defined constraints are used to apply additional logic in both validators. The constraints are used in the front end to notify the user about erroneous input before it is saved and sent to the back end. In the back end, the constraints are used to prevent erroneous data from being persisted in the database. Using OCL models ensures consistent handling of data in both the front end and back end.
MaCoCo uses the CDTagging-language (cf. Section 3.5.1) to link constraints to classes and attributes.

8.5.2 MaCoCo-Specific Extensions

As models defined in GUIDSL v1 can not be extended or modified with custom GUI components, the only alternative to writing use-case-specific user interfaces by hand is a corresponding grammar extension. The grammar of GUIDSL v1 is defined modular in order to enable the extension for specific use cases. Extension for specific use cases is enabled via an extension of the GUIDSL-grammar. As MaCoCo is one of the larger use cases that incorporates GUIDSL v1 models, an additional component grammar was added to incorporate MaCoCo-specific components.
The component grammar GUIDSLMaCoCo contains page elements that were developed specifically for the MaCoCo use case. The grammar defines the following elements:

- **`MacocoBalanceBox`**: Text field, styled to highlight the a current monetary value
- **`ExcelImport`**: Page Element that starts an Import Dialog for Excel files.
- **`CSVImport`**: Page Element that starts an Import Dialog for CSV files.

MaCoCoBalanceBox

A MaCoCo-specific component grammar was introduced in order to keep page elements that are very customized for a specific use case separated from other generic page elements. One example for such a specific component is the `MaCoCoBalanceBox` as shown in Listing 8.1. A balance box is defined by a reference to a parameter and a simple title.

```
MCG
MacocoBalanceBox implements PageElement =
  "balances" ref:Reference "{"
    Box+
  "}"
;

Box =  "box" (label:String) "," amount:Reference  ("," type:Name&)?  ";
    ";
```

Listing 8.1: Excerpt of the grammar *GUIDSLMacoco* defining a balance box page element

```
GUI-DSL v1
balances <bi {
  box "Remaining Budget", <bewilligungsBudget, primary;
  box "Planned Expenditure", <expenditure, danger;
  box "Expenditure", <planBudget, default;
  box "Total Budget", <restBudget, success;
}
```

Listing 8.2: Excerpt of a GUIDSL v1 model defining the balance box page element shown in Figure 8.18

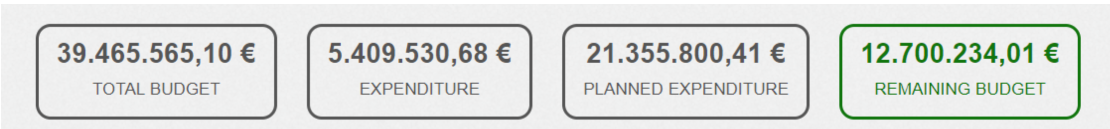


Figure 8.18: Example of a balance boxes as used in MaCoCo

8.6 Lessons Learned from the MaCoCo Project

MaCoCo has been in development for a long time, and a lot of the experience gained during the development of both the web application and the MontiGem-framework has already been incorporated into other model-driven projects. As summarized in [BGK$^+$23b] there are four key lessons we learned from developing MaCoCo with a model-driven approach:

1. *A model-driven approach is beneficial not only in the long term but also at the prototyping stage, especially when basic generation tools and experts in Model-Driven*

Engineering are on hand: The benefits of MDSE are well known and established [Rum12, Rum17, HKR21]. However, when initially setting up a prototype, extra effort is needed to implement and establish the generator infrastructure and build systems. Nevertheless, within the MaCoCo project, this effort paid off as the system gained additional flexibility that was needed to cope with rapidly changing requirements. Models could be changed easily resulting in corresponding consistent changes throughout the target code.

2. *To introduce features that impact the entire system, the generator can be extended. However, for more complex situations, additional Domain-Specific Languages might be needed:* As shown in Figure 8.3 languages were developed in order to increase the amount of generated code, and increase the benefits of a model-driven approach. Each model is to a certain degree an abstraction of a system that it describes. Therefore the domain-specific code we generate based on these models is limited by this abstraction. Within the MaCoCo use-case, we identified over time multiple aspects that could be generated without the need to add further details to the root class diagram *e.g.,* the persistence infrastructure can be generated with the same model as the command infrastructure by extending the generator. Data validation however does require a lot more information than basic data structures can provide, therefore we needed to add support for another DSL.

3. *For projects anticipated to undergo further development, it's crucial to integrate a generator early on*: MaCoCo is a fast-growing complex software project. Keeping the code quality and structure consistent turned out to be a challenge, especially in the front end of the application, as the used technology stack and the experience of each development team member did not always match up. This issue was solved by introducing the GUIDSL v1 and a corresponding generator infrastructure. The introduction of such an infrastructure would have been easier in an early stage and is almost infeasible at the current stage of the software.

4. *For a successful transition from the current use case to a general-purpose generator framework, it's essential to distinguish between generic and use-case-specific generator code:* After three years of development, the generator already produced many of the necessary features to create an information system automatically. However, these features were interwoven with use-case-specific elements of the MaCoCo project. Thus a lot of effort had to be put into the separation of generic and MaCoCo-related code in order to use the generator framework for other use cases than MaCoCo. A large portion of the generator had to be refactored, as this process covered all generated artifacts, including user interfaces, commands, persistence logic, and overall generator usage. Developing the MontiGem-framework as a generic framework from the beginning would have saved a lot of time and resources.

Chapter 9

Further Applications Studies

The generative approach was used and tested in multiple projects [BGK+23b]. In the following, we list a few and point out lessons we learned from each project.

Contents

9.1 Agile Data Dev - Data Management for Wind-Turbine Engineering

The Agile Data Dev (ADD) project aims for quicker market release and enhanced efficiency in wind turbine development. ADD was the first project utilizing a CD2GUI-based approach (*cf.* Figure 9.1). The MontiGem-framework was used to create a web-based information system that served as a central point of information exchange [MNN+22] for the client. The system was used to store development artifacts, such as CAD files, configuration parameters of the wind turbines, and corresponding simulation results. By providing this centralized single point of truth, data could be exchanged more efficient between employees, replacing previous approaches where engineers stored data on their individual computers. In addition, the generated application provided functionality to archive data in accordance with legal requirements, increasing the overall efficiency of the data management. As this project relies heavily on data entry and storage, it was as an ideal candidate to serve as a proof of concept for CD2GUI.

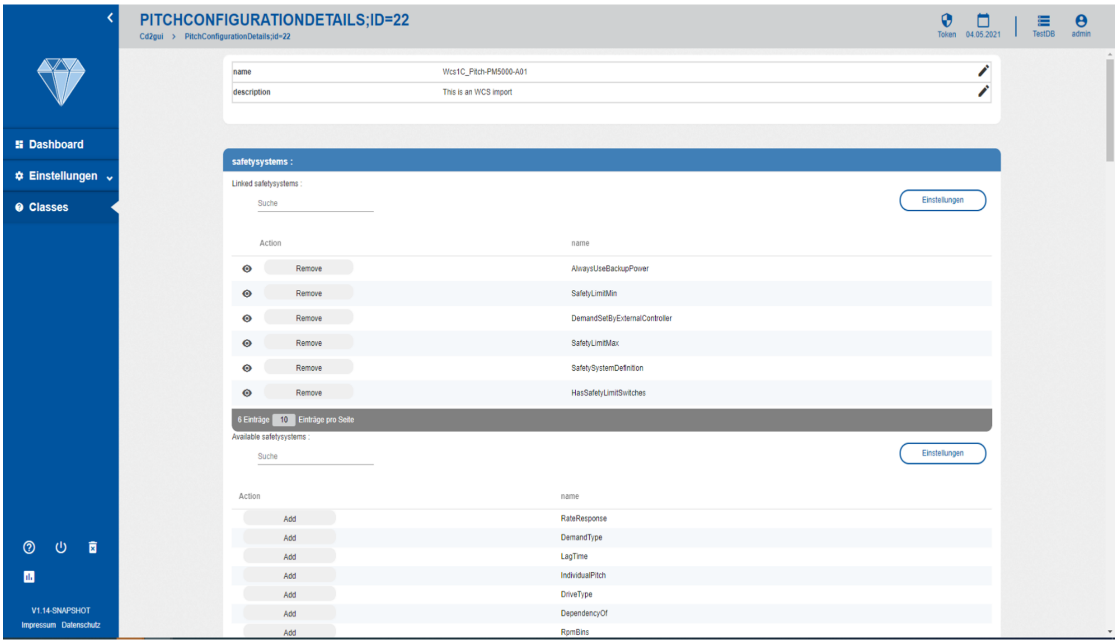


Figure 9.1: GUI provided by CD2GUI with MontiGem-framework for the ADD project showing the details page of a 'PitchConfiguration' object with ID 22. Next to the name and the description of the component, linked safety systems and available safety systems are shown.

9.1.1 Evaluation and Lessons Learned

This particular project benefited greatly from its data-centric approach. We were able to apply our methodology to quickly deliver a fully functional application prototype and iteratively improve the application at an early stage, as the majority of the required data structure was already known.

Need for Extensibility and Customization

Although we were able to provide an approach that created an application to manage all provided data, the end users required adaptations of specific data fields to be more productive. In addition, further logic was requested besides the basic data retrieval and entry, which CD2GUI provides by default. To be more flexible for future projects, both GUIDSL and CD2GUI had to be adapted to introduce adaptability and variability to the methodology.

Additinal Views for Data Strucrtures

As the project relied heavily on the provided data structure, the client wished for direct visualization of the used class diagram (single underlying model). Thus a diagram-based navigation was developed (*cf.* Section 6.3.2). In addition, the LLM4CD tool also provides a visual representation of any class diagram created.

Live Prototypes Drive MDE Toolchain Development

While the MontiGem-framewok and CD2GUI could provide a fully functioning prototype, the system had to be extended rapidly, to keep up with the client's feature requests, pushing the limits of both the generator framework as well as the used DSLs. Based on the given feedback many extensions were implemented and in addition the gained expertise was used to develop a new modeling language for user interfaces: GUIDSL v2 (*cf.* Section 5.2). Hence one of the distinguishing aspects of both languages is the significantly improved, extensibility and adaptability of GUIDSL v2.

9.2 InviDas - Interactive, Visual Data Rooms for Sovereign, Data Protection Decision-making

Interactive visual data rooms for sovereign decision finding regarding data protection (InviDas[1]). This project (*cf.* Figure 9.2) focuses on visualizing the privacy policies of smart wearables and aims to facilitate comparisons between different devices. It receives funding from the German government through the *Bundesminiterium für Bildung und Forschung* (BMBF) and is brought to life by a collaborative effort of software engineers, human-computer interaction experts, and smart wearable producers.

Software development started after MontiGem has already been in use for various projects. Consequently, the enhancements made to MontiGem during this time were primarily centered around resolving minor bugs rather than adding new features. For instance, issues such as the unnecessary redirection of anonymous users to a login page, even when a user login was not required to access certain parts of the website, were addressed. Importantly, these issues were confined to MontiGem's runtime environment and did not impact the generator itself.

[1] `https://invidas.gi.de/`

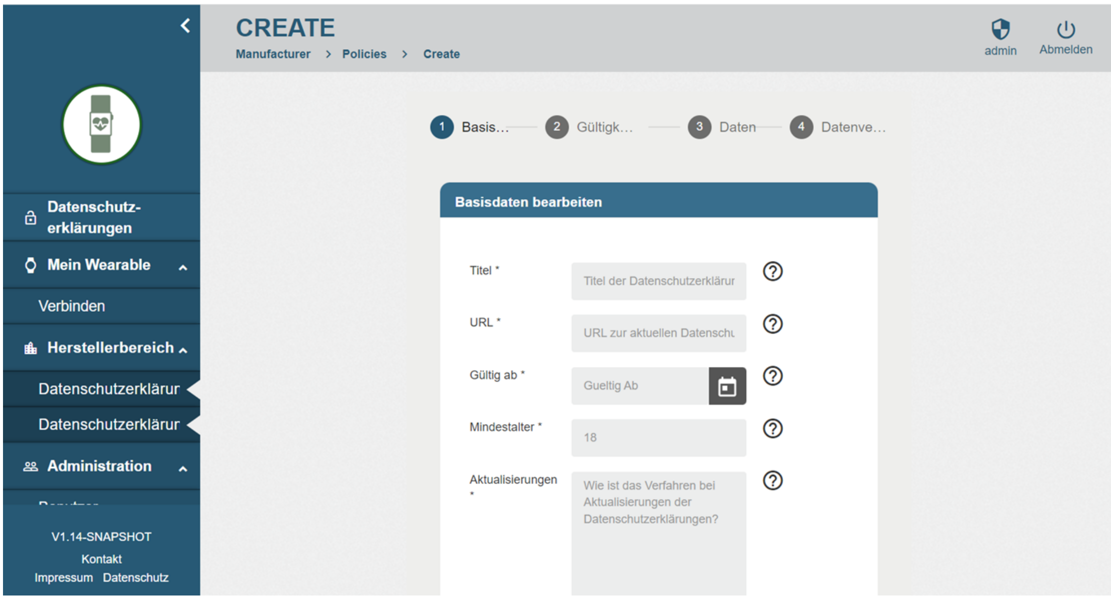


Figure 9.2: GUI provided by MontiGem-framework for the Invidas project showing a user interface for user data entry

9.2.1 Evaluation and Lessons Learned

In contrast to previous projects InviDas aimed to use MontiGem as-is and deploy it within the project's use case. The InviDas project had created a web application that enabled users of smart wearables to easily understand and compare manufacturers' privacy policies through data points instead of legal text. This approach was based on data processing categories derived from analyzing various smart wearable privacy policies. The platform aimed for manufacturers to directly input their privacy policies. A significant innovation of the project was the development of a standard privacy policy model, created by analyzing the privacy policies of seven major smart wearable vendors in accordance with the GDPR [BCC+22]. This model served as the main development artifact for the InviDas platform, facilitating the generation of the back-end structure through MontiGem.

Handwritten extensions to generated code are an imperative feature

Similar to MaCoCo the Invidas use case required several features that were not covered by the default generated code of the MontiGem-framework. A modification or extension of the generator was unfeasible as the required modifications for the features would not be systematic and only occurred in low numbers throughout the target application. Thus, additional features were primarily added via handwritten extensions to the generated source code (*cf.* TOP-Mechanism Section 7.7)

Prototyping accelerates communication and thus software development

Translating unstructured data, such as legal texts processed in InviDas, into structured formats like class diagrams (CD4A) presents significant challenges, especially as often not all stakeholders are familiar with either legal texts or class diagrams. This necessitates extensive domain analysis and ongoing communication between the stakeholders and often results in frequent system modifications, especially in the early stages of complex projects. Generators like MontiGem prove invaluable in these scenarios, enabling rapid implementation and deployment of changes to the data model as additional domain knowledge is acquired over time.

Prototyping helps to identify problems early

In the initial phases of software development, concepts often lack maturity, and their testability can be challenging, making it difficult to identify and address potential issues early on. Prototypes serve as a crucial tool in this context, allowing for the testing of use cases at a very preliminary stage. By doing so, they help to save valuable time that would otherwise be spent developing these underdeveloped concepts and subsequently fixing them. This early-stage testing ensures a more efficient use of resources, leading to a smoother development process and a more robust final product.

9.3 Ford Pro Tool Tracking Platform

In collaboration with Ford Research and Innovation Center Aachen, we successfully completed a pilot project for Ford Pro, aimed at helping companies monitor their tools and machinery using Bluetooth, AI, and GPS tracking (*cf.* Figure 9.3). This system reminds manufacturers driving the company vehicle if a tool is missing from their vehicle, potentially left behind at a job site, and allows companies to locate and manage their equipment in real-time. This can result in significant savings, estimated at €38 per vehicle monthly, and time efficiency, saving workers approximately one hour and administrators nearly two hours per week.

9.3.1 Evaluation and Lessons Learned

MontiGem was used to generate the corresponding web application. This project introduced new use cases to the generator framework as not only the support for common web browsers but also support for mobile devices and infotainment systems of the vehicles were required.

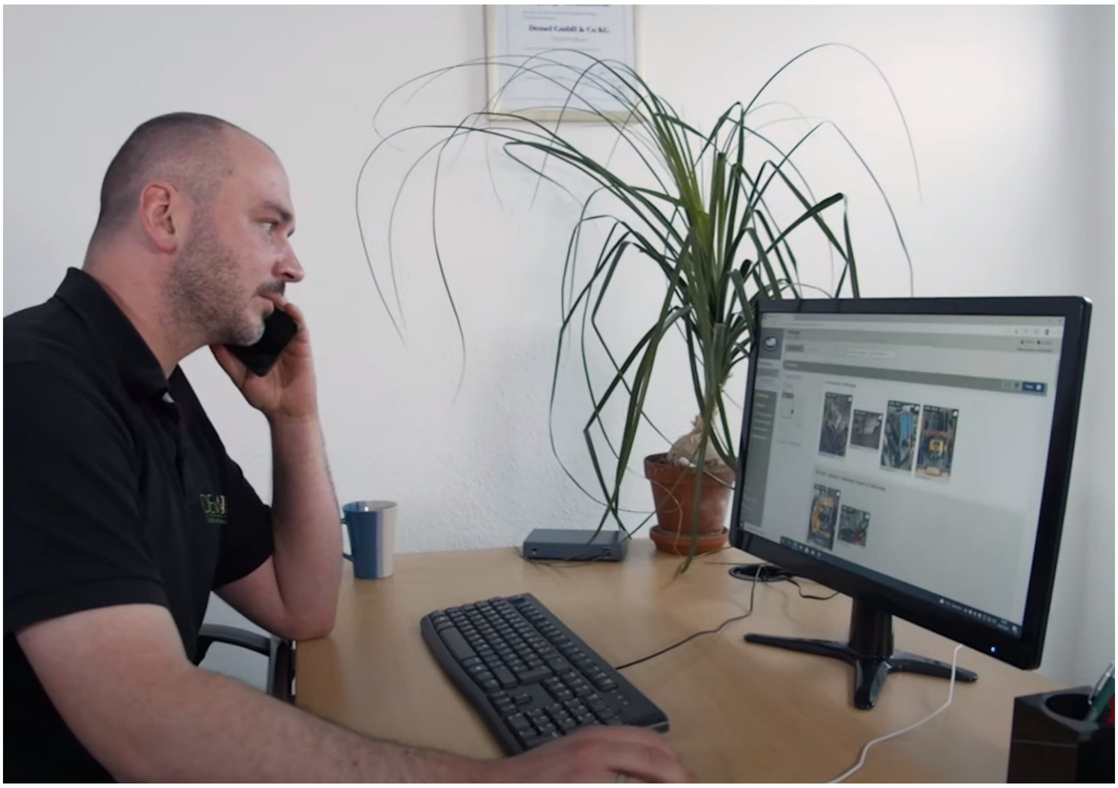


Figure 9.3: Emlpoyee monitoring tracked tools with the MontiGem user interface. The image shows a still frame from Ford press release: `www.youtube.com/watch?v=ODuvZ6AahzI` (accessed 1.12.2023)

Models serve as Effective Communication Aids

The development of the application involved a large variety of stakeholders from different disciplines. Developing a common understanding of the envisioned application is key to an efficient development process. Throughout development, the class diagram turned out to provide an ideal foundation to communicate the current state and upcoming changes to the application.

Identify changes to the generator architecture early

In our MontiGem-based application, we developed mobile versions for smartphones and a tablet specifically designed for vehicle use. The tablet version required a substantial redesign of the user interface to prevent driver distraction, achieved by limiting information displayed based on the vehicle's acceleration. However, the application faced challenges with unreliable mobile connectivity and a less responsive user experience compared to native apps. Consequently, we decided to phase out the web version and transition to a native Android app in subsequent releases. Identifying these problems early on could have led us to generate an Android app from the beginning, saving resources and effort spent on the web version of the application.

9.4 Usage as a LCDP in Teaching

Since the approach described in this work is suitable for demonstrating the reduction of code to be implemented through the use of model-driven software development, the approach was also used for teaching. The approach was evaluated in the context of a student lab for 14 weeks. Within the lab, the MontiGem-based approach was evaluated next to an A12-based approach. As both systems are intended to generate web applications but rely on different DSLs to do so, the lab was used to evaluate modeling methodologies and the performance of both frameworks. Students had to develop an application for the same use case with each platform. Half of the students started using A12 the other half started using the MontiGem-based approach to reduce an experience-based bias.

9.4.1 The A12 LCDP

The enterprise LCDP A12 [mtpG21] is developed by *mgm technology partners* and used in customer projects for the application domains insurance, e-commerce and the public sector. Applications realized with A12 are typically single-page web applications, based on a client-server architecture. Software development within A12 is divided into two parallelizable branches.

9.4.2 The Application Modeling Process

The *application modeler* begins by:

1. Installing modeling tools using the provided installer.
2. Utilizing visual model editors to define domain-specific data and UI models (see Figure 9.4).
3. Formulating domain-specific constraints using a Domain-Specific Language (DSL). This generates validation code for runtime data consistency checks.

Given A12's emphasis on processing forms and documents in enterprise business applications, these elements are particularly prioritized. In UI models, data models are linked and mapped to UI components. These components, or widgets (e.g., a data picker), are orchestrated to render the UI as modeled (*cf.* [mtpG20]). The engines use both the validation code and UI models to highlight incorrect entries.

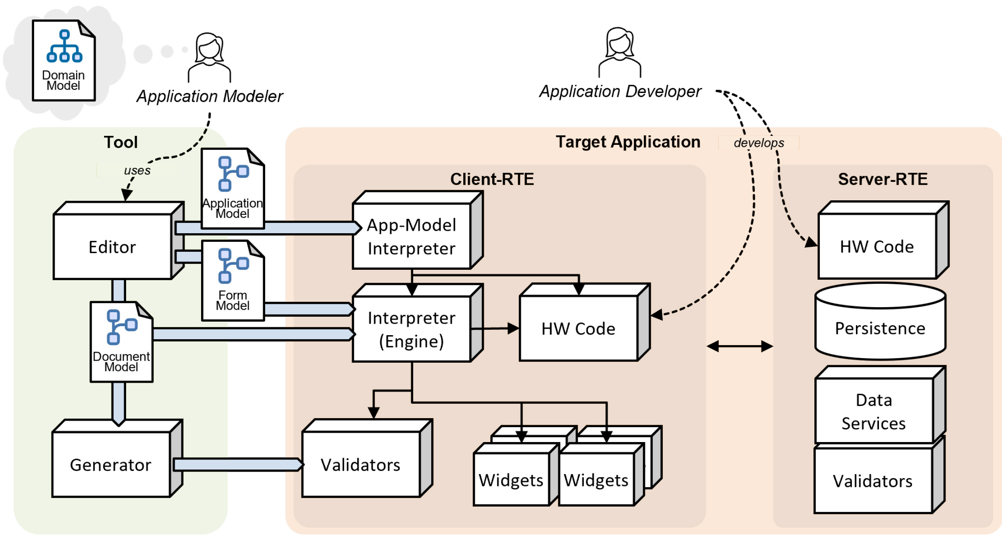


Figure 9.4: basic A12 architecture

The *application model* dictates the display of UI components, defining both the UI layout and behavior. For instance, it may specify a tabular data overview on the left and corresponding record details on the right. Post-modeling, these models are deployed to the target application via the modeling tool.
In parallel, the *application programmer* begins with a provided template, serving as the development starting point. Their primary role involves registering the UI components, as referenced in the application model, within this template.
MontiGem and A12 share many elements in their architecture as both platforms operate model-driven and target the generation of a web application: *cf.* Figure 9.5 and *cf.* Figure 9.4.

9.4.3 Results and Lessons Learned

The student lab improved the product maturity, by identifying multiple edge-cases and by pointing at gaps in its documentation. Each student group developed a running web application (*cf.* 9.6(a), 9.6(b)). A follow-up questionnaire showed that the MontiGem-based approach used within the limited scope of the lab, was able keep up with an LCDP approach from industry. Students noted little to no drawbacks from using textual models, but indicated limitations by being constrained, to the limited set of GUI components, and lack of extensibility of GUIDSL v1. This feedback was considered in the development of GUIDSL v2. Although mentioning limitations, all students noted that they were able to develop the intended application, with the exception of minor stylistic elements or features that would have improved the comfort of using the application.

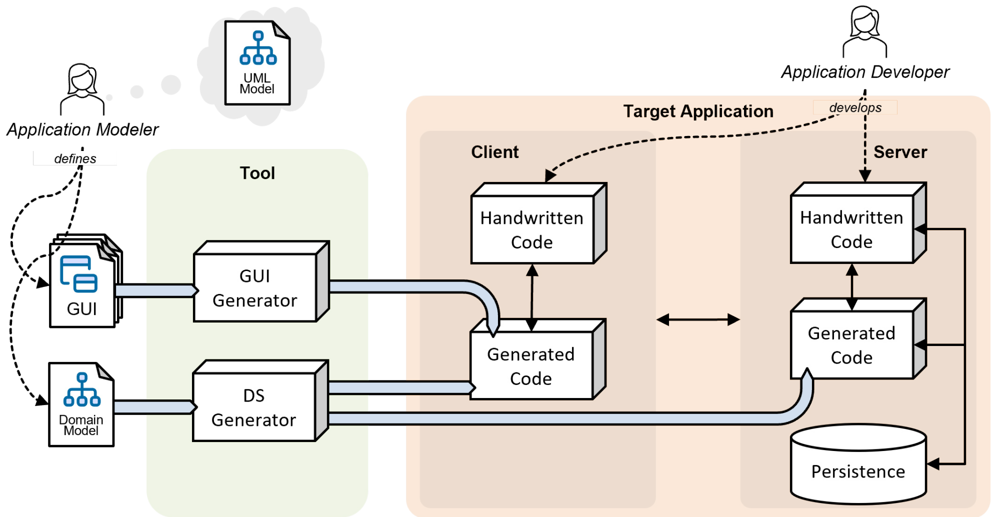
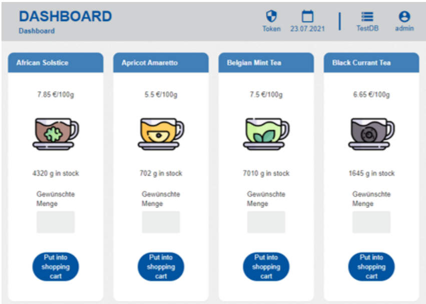


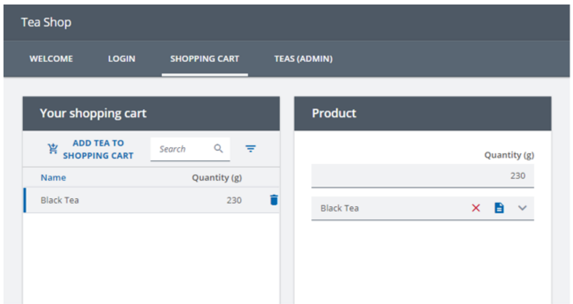
Figure 9.5: Basic MontiGem Architecture in comparison

9.5 Consulting Use Case

In order to evaluate the produced application prototype, an application was built using two different methodologies. The targeted application is an information system that handles document management for a consulting firm, it contains simple data entry and retrieval, as well as low-level security requirements and a document storage feature that MontiGem does not provide in its default configuration. An experienced software developer used both the MontiGem-based approach and a set of commercial frameworks to build the same application twice. Both applications were presented to stakeholders of the consulting firm and potential end users in order to evaluate the application. The usability of the developed applications was evaluated through a user study. Participants were given tasks to perform using the applications, and their interactions and feedback were observed and recorded. The study aimed to identify any obstacles or difficulties faced by the participants while using the applications. The feedback and observations from the user study were used to assess the usability of the applications.
As expected the results of the user study showed that the custom web application performed better in terms of usability compared to the MontiGEM-based application. Participants found the custom application easier to use, were able to navigate it more effectively, and found it more satisfying to use. They also rated the aesthetics of the user interface higher in the custom application. However, there were some usability issues identified in both applications, such as unclear UI elements and a complex navigation bar in the MontiGEM-based application. Overall, the custom application received better feedback in terms of usability.



(a) Screenshot of a generated web-application using the MontiGem-based Approach



(b) Screenshot of a generated web-application using the A12-based Approach

Figure 9.6: A web store for tea developed with both A12 and MontiGem.

This result is expected as a MontiGem-based prototype is very likely to have a low TRL-Level compared to a commercial solution. An interesting aspect of the user study is that despite the prototype nature of the MontiGem-based application, its evaluation for each aspect remained above 5 on a scale from 0 to 10 (positive tendency).
The performance of the developed applications was evaluated using multiple methods. One method was the Lighthouse evaluations, which focused on the frontend's performance. The Lighthouse results measured page loading times using metrics such as Speed Index (SI) and Largest Contentful Paint (LCP). The custom application received a Lighthouse score of 99/100, while the MontiGem-based application received a score of 76/100. The custom-built version of the application had an LCP of 909 milliseconds on average, which is considered very good. On the other hand, the MontiGem-based version had an LCP of 2983 milliseconds on average, which is below the 50th percentile and considered bad. The Speed Index measurements also showed that the custom version had an average value of 243 milliseconds (excellent), while the MontiGem version had an average value of 1803 milliseconds (average). Overall, the custom version performed better in terms of loading times compared to the MontiGem version.

9.6 Fenix / MontiGem 3

MontiGem has been developed and optimized over multiple years. As a result, a new version of a generator (Fenix) was developed using up-to-date frameworks and incorporating many of the lessons learned from MontiGem 1 (*cf.* [BGK+23b]).
The architecture of Fenix (*cf.* Figure 9.7) is similar to MontiGem (*cf.* Figure 7.1). The new generator uses GUIDSL v2 and CD4A as input. Similar to MontiGem the generator produces a complete web application. In contrast to MontiGem Fenix generates Java for

Question	**Model Driven**	**Custom**
The application was easy to use.	7	8.6
I found my way around the application well.	6.2	8.9
The application was satisfying to use.	5.9	8.8
I liked the application from the aesthetics of the user interface.	6	8.7
The user interface is structured in such a way that I can easily find my way around it.	5.9	8.6
You need a (short) learning phase to be able to use the application smoothly and comfortably. With 0 being no learning phase and 10 being a long learning phase.	5.3	3.1
The loading times of the application were short and pleasant.	7.6	9.4
The overall impression of the application was good.	6.6	9.2

Table 9.1: Average results for all aspects of the user study (Higher is better). Ten users were provided with both implementations of the application.

the presentation tier, which is transformed into TypeScript. Providing a consistent Java code base. A key difference between MontiGem and Fenix lies in how the data structure is shared between the three layers: the presentation layer, the application layer, and the persistence layer. MontiGem shares the same data structure in the persistence and application layer. The presentation layer presents a 'view' of those layers. That can be either be reduced or extended data structure providing additional parameters e.g. in case computed values are required for the UI or a reduced data structure in case values have to be omitted, such as passwords. In the Fenix architecture, the presentation layer and the application layer share the same data structure. The data structure handled by the presentation layer is identical to the data structure handled by the application layer - allowing for better synchronization between multiple clients and the server. However, not all data within this data structure is persisted thus the data structure from the persistence layer might differ. Parameters that should not be persisted are defined in the data structure as derived attributes or classes and often contain computed parameters and temporary values.
SEHub[2] is a platform for software project management, including simple issue management and artifact analysis. The application consists of four major Features:

- **User Management** The application implements both authorization and authentication. User management is supported, adding and removing users as well as granting roles and permissions.

- **Issue Management** Issues can be created having a progress status. Each issue can

[2]`https://sehub.demos.se.rwth-aachen.de/gui/Start` (last accessed 1.12.2023)

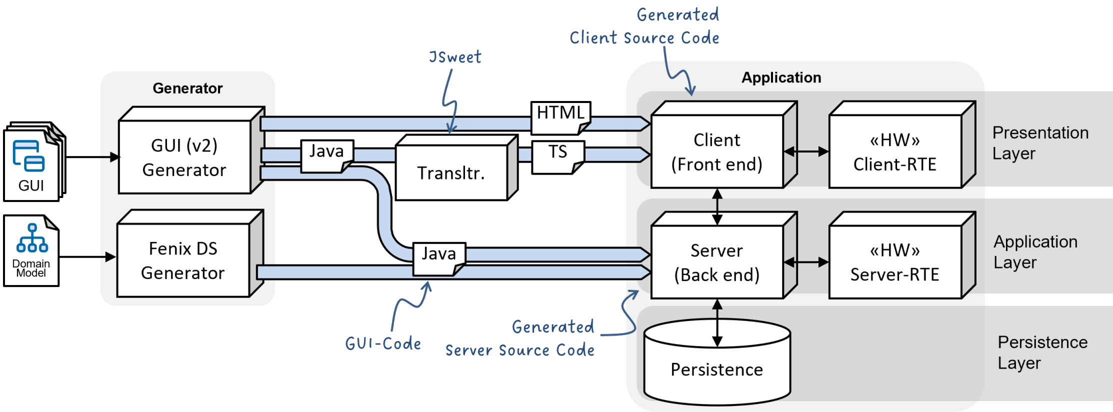


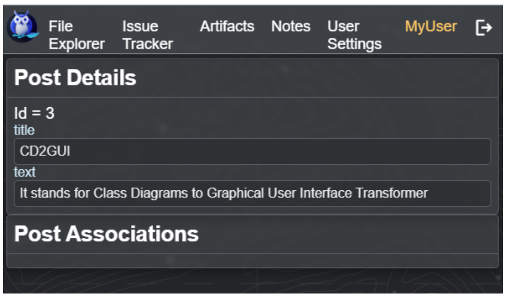
Figure 9.7: Fenix architecture: User Interface (GUI) Generator and Data structure (DS) generator primarily producing Java and transpiling to TypeScript later

have its own commentary thread that is coupled to user management, enabling the users to have issue-related discussions.

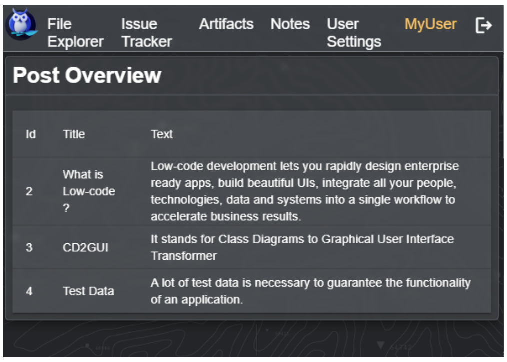
- **Artifact Management** SeHub supports artifact analysis, enabling the browsing of project-related repositories and dynamic visualization of all project-related artifacts and their relationships.

- **Project Management** Projects can be managed by linking issues, comments, and artifacts to individual projects and providing access management for each.

The application was developed as a proof of concept of the architecture. CD2GUI was used to provide initial user interfaces and provide a prototype of the concept. In addition, CD2GUI proved useful to test newly defined user interfaces and set up test data (*cf.* Details Page 9.8(a), Overview Page 9.8(b)). As CD2GUI provides all pages necessary for data entry, it could be used to add and remove data while modeling new interfaces. In general, the approach operated as intended and provided support in rapidly developing early prototypes. However, as the application itself, remained in a prototype state, the capabilities to transition to a real-world application could not be demonstrated in full within this project.



(a) Details-Page



(b) Overview-Page

Figure 9.8: Pages generated with CD2GUI for the MontiGem 3-Framework

Chapter 10

Discussion of the Approach

In the following, we will discuss the approach as a whole and take a closer look at the individual elements of the toolchain. We will compare the methodology with other common approaches from the industry such as Low-Code Development systems.

Contents

10.1 Methodology

The presented toolchain is assembled by three consecutive transformations: LLM4CD, CD2GUI, and MontiGem. We will discuss each transformer individually.

10.1.1 Using LLMs to transform natural Language into Domain Models

Model-driven software development is a powerful paradigm to efficiently design and create software. Although it can reduce the development time in the software engineering process dramatically it also introduces high amounts of computation power making the entire approach less sustainable [DBK23]. AI-driven Software engineering tackles the complexity of software development by filling the gap between domain experts and software engineers [FR07, FL10]. Though smaller, the gap still exists. Among the challenges the domain expert has to learn when creating a new model in a new DSL are [BAGB14]:

- Syntax and semantics of the language.
- How to compose the syntax to perform a function.
- Comprehension of syntax written by others.
- Debugging of syntax.
- Modification of a model.

Recognizing this issue, several approaches attempt to use structured text [ENT15] or natural language [YS22] as input to produce corresponding UMLP diagrams. A survey from 2021 [AMH21] analyzed 24 tools and methods that use rule-based or statistical methods to attempt the extraction of UML class diagrams from natural language specifications. It concludes that there is no tool or method that is able to automatically generate complex UML class diagrams. For this reason, the usage of language models was evaluated to tackle this challenge. Next to our own findings (presented in Chapter 4), there is further research investigating the capabilities of LLMs to facilitate the automatic generation of domain models: [FFK23, CYC^{+}23]. Camara et al. [CTBV23] assess the capabilities of AI in modeling tasks and conclude that in general, ChatGPT is capable of producing syntactically correct UMLP models and point out the necessity to integrate generative AI stronger into model-based software engineering processes.
We can conclude that LLMs are currently among the best options to transform natural language into a UMLP model.

10.1.2 Transforming the domain model into GUI-models

LLMs are well suited to produce models of generally known modeling languages (*cf.* Chapter 4), however, the performance strongly declines in case the LLM is tasked with the generation of uncommon or not yet published LLMs. As we need to close the gap between natural language and PSMs for the web application generator, we use the well-working creation of domain models and transform these into the corresponding user interface models using the M2M-transformer LLM4CD.

As we need a consistent set of models for our system the LLM-based approach might not be the best choice to produce GUI-models directly. All generated overview pages and details pages should look and feel the same to provide a consistent user experience. The natural randomness that heuristic approaches introduce in this transformation is in this case a disadvantage that is hard to avoid.

10.1.3 Generating an Information System

The benefits of model-driven software engineering are well-established [Som11, Rum17, CFJ+16, Lid11]. Hence the methodology is set up to use MDSE in order to produce software. The preceding two transformers result from the goal of reducing the problem implementation gap [FR07] while still remaining model-driven. There are multiple approaches also using model-driven approaches to generate web applications [TMN+04, FP00, ADB13, BDLD11]. As we have also published approaches to generate web applications using our tooling [ANV+18, Rei16], we use and develop for this methodology DSLs transformers and generators that all are based on the same language workbench (MontiCore) instead of adopting new DSLs that are based on multiple different language workbenches and technology stacks thus leading to a more complex and harder to maintain toolchain.

10.2 Software Engineering with Low-Code Development Platforms

There are several definitions for low-code development platforms (LCDPs), but, in general, they have in common the reduction of the need for handwritten source code by using a form of high-level abstraction to define an application. Taking this into account, the methodology presented in this thesis and LCDPs share similar goals and solve similar challenges.

10.2.1 Definition

The evolution of software development over the recent decades has always been accompanied by the aspiration to reduce the amount of handcrafted code required to produce software [DRKdL+22]. In the 1980s, tools like the Fourth Generation programming language and CASE tools [Mar82] should perform this task. In the 1990s the 'Rapid Application Development' emerged [Mar91], and in the 2000s the 'End user Development' approach [LPKW06]. Up until today, Model-Driven Development(MDE) is a popular approach to tackle complex implementation problems [MCF03].
The concept of *Low-Code* was initially introduced in a market analysis conducted by Forrester in 2014 [RRM+14], in which such platforms were characterized as:

Definition 11. ***(Forrester 2014)*** *platforms that enable rapid delivery of business applications with a minimum of hand-coding and minimal upfront investment in setup, training, and development. [RRM+14]*

Here Forrester refers to LCDPs as a developing methodology that reduces the amount of required handwritten code, referring to the reduction of implementation as 'low code.' Since this initial characterization of LCDP, multiple different definitions for the term occurred. Depending on the context and use case LCDP are defined differently. In 2017 Forrester defines LCDPs as:

Definition 12. ***(Forrester 2017)*** *products and/or cloud services for application development that employ visual, declarative techniques instead of programming and are available to customers at low- or no-cost in money training time to begin, with costs rising in proportion of the business value of the platforms. [Ho16]*

We observe an increased focus on platform aspect and its tooling to not only reduce the amount of required hand-crafted code, but also an emphasis on visual modeling [RKL+19]. Similar to Forrester, Gartner identifies platforms that support the development of software. In [VID+19] Gartner introduces low-code application platforms (LCAPs). These are a special case of LCDP that aim to develop enterprise-class applications, fulfilling requirements such as high performance, scalability, high availability, security, resource use tracking and API access to and from local and cloud services [DRKdL+22].

Definition 13. ***(Gartner 2019)*** *A low-code application platform (LCAP) is an application platform that supports rapid application development, one-step deployment, execution and management using declarative, high-level programming abstractions, such as model-driven and metadata-based programming languages. They support the development of user interfaces, business logic, and data services, and improve productivity at the expense of portability across vendors, as compared with conventional application platforms. [VID+19]*

Since its discovery, multiple companies have developed and sold LCDPs [RA17]. A Rymer market analysis identified 13 platforms in 2017, labeling Appian[1], Kony[2], Mendix[3], OutSystems[4] and Salesforce[5] as leaders in the segment. With a market valuation of $2 billion for Appian, investments of $360 million in OutSystems and Siemens

[1] https://appian.com/
[2] https://www.kony.com/
[3] https://www.mendix.com/
[4] https://www.outsystems.com/
[5] https://www.salesforce.com/

by Mendix for $730 million, Forrester estimates a global market size for LCDPs of $3.8 billion.
Platforms that require no implementation of source code by the developer in contrast to a low amount are called *No-Code Development Platforms (NCDP)* or sometimes also *Zero-Code Development Platforms*. NCDPs are designed to meet the basic needs for the development of a system. The lack of coding can limit the complexity in platform interactions and thus limit the complexity of the resulting application. NCDPs are therefore well suited for simple small-scale applications that need a short development time but lack the adaptability and flexibility that LCDPs provide.

Definition 14. *A No-Code Development platform is a Development Platform (NCDP) for Web-based Applications, that requires no hand-written source code or any Software development expertise to produce a running Software system. NCDPs typically provide visual editors to ease the definition of the desired target app.*

In addition to LCDPs, multiple NCDP, are being developed and used commercially. Known platforms are Canva [6], Appsheet[7] (Recently aquired by Google) or ClickUp[8]. The following LCDPs are commonly used in industry:

10.2.2 Mendix

Mendix is a low-code application development platform that allows users to design, build, test, and deploy web and mobile applications with minimal manual coding. It provides a visual development environment, pre-built templates, and drag-and-drop components, making it accessible to both professional developers and nontechnical users.
Mendix is designed to accelerate the application development process, reducing time to market while maintaining the ability to create robust, secure, and scalable applications. It offers features like multi-device support, integration with existing systems, and a collaborative environment for developers and stakeholders.
Mendix is used across various industries and sectors to develop enterprise applications for purposes such as process automation, customer engagement, and digital transformation.

10.2.3 OutSystems

OutSystems is a low-code platform that enables users to develop, deploy, and manage custom enterprise applications rapidly. Its designed to simplify the application development process through a visual interface, drag-and-drop components, and pre-built templates. This approach significantly reduces the amount of manual coding required, making it accessible to both professional developers and non-technical users.

[6] https://www.canva.com/
[7] https://www.appsheet.com/
[8] https://clickup.com/

OutSystems offers a wide range of functionalities, including integration with existing systems, support for multiple devices and platforms, and tools for automating application life cycle management tasks. It is particularly popular for its ability to create scalable, secure, and high-performance applications while accelerating development timelines. The platform is suitable for building various types of applications, such as web, mobile, and progressive web applications (PWAs), catering to different industries and business needs. In the context of this work, we will refer to a LCDP as a platform as defined in Definition 12. The MontiGem platform relies on the definition of textual models, rather than designing the models in a visual editor. The integration of a visual editor was evaluated, but the increased development effort does not pay off compared to the advantage of defining models in the platform itself. Textual models can easily edited in the same development environment, the hand-crafted code is developed in. Thus, although the platform does not have a visual editing component we still refer to it as a low-code platform as it still reduces the required hand-written code.

10.3 Software Engineering with Large Language Models

Zhou et al. [HZL+23] give a detailed overview on the current state of LLM-based software engineering. Recent advances in the development of neural networks and language models have led to several potential breakthroughs for software engineering tasks.

- Synthesize complex code [CLS17]
- Summarize source code using neural networks [ZWZ+20]
- Using LLMs to find and repair software vulnerabilities [CTJ+23].
- Repairing JavaScript programs using GPT-2 [LCV22]
- Using ChatGPT as a programming assistant [TLL+23]

Siddiq et al. [SCS23] describe a lightweight framework to generate and evaluate code snippets for Java and Python. The approach describes a toolchain that uses several filters and rankings to ensure the quality of the code produced. Although there are several approaches using language models to support the developer in software engineering tasks, these approaches have the following challenges in common. The software engineering domain is a rapidly evolving field, to keep up, language models have to be updated regularly. A language model will most likely not know any new API definitions or consider newly discovered security issues in a similar capability to a human developer [WZK+23]. Language models are statistical models; therefore, the suggestions they provide are based on the statistical distribution of data the model is trained on. Thus, the outcome is strongly dependent on a well-chosen input data set and will only provide the most likely data: For example, it will not choose the best solution, but rather the

most statistically fitting one used for a given input data set. As the LLM is trained on a lot of generic source code and is not trained on specifically well-written implementation the code it will synthesize will also be 'generic', as LLMs follow the *garbage-in-garbage-out* principle [KR18]. On the other hand, LLMs suffer from the *out-of-distribution generalization* problem [FGH+23]. The most common programming solution might not always be the best for a given problem, there is a large quantity of training data with low likelihood to be chosen, as they were applied only a few times in the training data as well. [ZKX+23] finds that software engineering data generally follows a long-tailed distribution, where a small number of classes have a large number of samples, while many classes have very few samples. This skewed distribution significantly impacts the effectiveness of LLMs for code, with these models performing between 30% and 254% worse on infrequent labels compared to frequent labels.
In summary, LLM-based software engineering is an emerging field that undergoes rapid developments and breakthroughs at the moment. There is not yet an approach that facilitates the complete development of a complex application based on informal specifications. There is a need for an approach to more collaborative and integrated software engineering with LLMs [Lo23].

10.4 Software Generation with Large Language Models

The term *generator* is used with different meanings in the literature. In the context of MontiCore-based systems, we follow the architecture explained in the foundation chapter (*cf.* Chapter 3). In the context of machine learning, however, the term generator is often used in a more general sense as a tool that can produce text.
There exist many programming assistants that are often titled as code *generators*. Some of the more popular ones are: GitHubs Copilot[9], Stack Overflows overflowAI[10], Amazons assistant codewhisperer[11], tabnine[12] and IntelliJs coding assistant 'AI assistant'[13]. These assistants are well suited to serve as pair programmers [WK+23, VZG22] however, they are not intended to produce software on their own and still require supervision by an experienced software engineer [FGH+23]. Hence we can not provide a programming assistant to a domain expert in order to enable him to generate an Information System.

[9] https://github.com/features/copilot
[10] https://stackoverflow.blog/2023/07/27/announcing-overflowai
[11] https://aws.amazon.com/de/codewhisperer
[12] https://www.tabnine.com/
[13] https://plugins.jetbrains.com/plugin/22282-ai-assistant

10.5 MontiCore-Based Systems

There are already a number of model-driven approaches that use MontiCore. In the following we look at the common intersections of these projects.

10.5.1 MontiDEx

In his thesis *Adaptable Code Generation of Consistent and Customizable Data-Centric Applications with MontiDEx* [Rot17] Roth describes the generator-framework MontiDEx. Similar to MontiGem it targets data-centric information systems and serves as a predecessor to it. As such it focuses on similar goals in its design:
MontiDEx uses model-driven development to reduce development costs by processing platform-independent models as primary artifacts. For the development of the framework, Roth developed and reused existing DSLs such as CD4A and the activity-diagram DSL: ADJava [Rot17]. He uses these DSLs for high-level abstractions and faces similar challenges such as the handling of the resulting underspecification while producing a functioning prototype. Roth tackles these problems by introducing *Hot Spots*: A specific code element that can be extended or modified. MontiDEx also supports the TOP-Mechanism [HKR21, DJR22].
In contrast to MontiGem, MontiDEx does not provide a web application. The MontiDEx Code Generator (*cf.* Figure 10.1) generates a desktop client that can communicate with a hand-written application server, whereas MontiGem generates the complete client and server architecture. As both approaches are data-centric MontiGem and MontiDEx use CD4A models as input artifacts, however, MontiDEx uses activity diagrams to define behavior whereas MontiGem relies on OCL and hand-written implementation of business logic. MontiGem can process GUI-models in order to define custom user interfaces, providing the ability to be adapted to a large variety of use cases, whereas MontiDEx only provides tabular user interfaces and entry masks in order to manage its data. A realization of MaCoCo would not have been possible with MontiDEx, for a variety of reasons some of which are:

- **Deployment:** A native desktop application requires considerations for each platform it runs on. As a browser-based application MaCoCo covers mobile, Windows, Apple, and Linux systems without additional implementation effort.

- **GUI:** A considerable amount of resources of a real-world system is invested in its user interface and the resulting user experience. MontiDEx does not provide model-driven support to implement these user interfaces.

- **Server-Client-Consistency** As further described in [GHK+20], a great benefit of the MontiGem-based approach is the resulting consistency of both server and client. As both are based on the same data structure, there is no additional effort required to ensure consistency. In addition, as the client is web-based, no version management or client-side update is required, as the user always accesses the latest version of the application. A desktop application would require active updating.

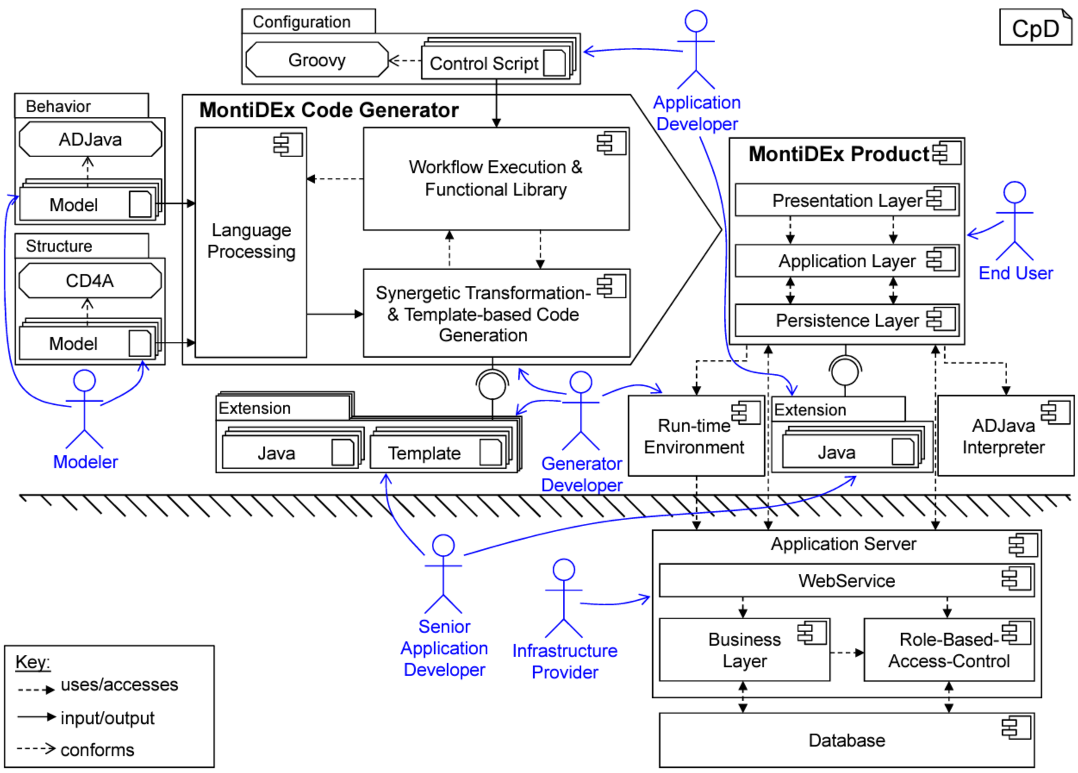


Figure 10.1: Architecture of MontiDEx by Roth [Rot17]

10.5.2 MontiWis

After the development of MontiDEx, Reiß developed a MontiCore-based generator for Web-based Information Systems [Rei16] (*Modellgetriebene generative Entwicklung von Web-Informationssystemen*). The work of Reiß builds upon MontiDex. The generator framework uses class diagrams (CD4A) and activity diagrams as input and introduces a new DSL to describe each generated web page, that serves as the direct predecessor for GUIDSL. As Reiß' UI-description language works in conjunction with the activity diagrams, and the IS we develop is a purely data-centric one, GUIDSL had to be newly developed.

MontiGem simplifies the development process by omitting activity diagrams from the generation process and using an architecture that puts a stronger emphasis on well-defined user interfaces and efficient data management of the system. In addition, MontiGem relies on popular frameworks such as Angular and Hibernate in order to optimize performance and reliability. The approach from Reiß relies on three kinds of models to define the Information system. In this work, we provide a methodology that still is capable of processing multiple kinds of models but makes the definition of such optional.

10.5.3 MontiTrans

In her thesis [Höl18], Hölldobler developed a framework (MontiTrans) to define transformations for a given DSL. The derived target-DSL specific transformation language could be used to apply endogenous transformations upon a model of the targeted DSL. Considering the motivation for CD2GUI, a transformation of models from one DSL to another, this approach solves many of the challenges CD2GUI had to face, however, MontiTrans was developed to primarily execute *endogenous transformations* (input and output models are based on the same metamodel), whereas CD2GUI provides an exogenous transformation (input and output models are based on different metamodels). The presented approach from Hölldobler is capable of the following:

- A process to derive a domain-specific transformation language for a given domain-specific language.
- Support for hand-written adaptation extension of generated transformation languages.
- Transformation libraries for MontiArc and CD4A models.

Although the MontiTrans approach had the potential to overcome the challenges of the CD2GUI, there were some reasons not to use the approach: First, the architecture was developed for an older technology stack and would have required expensive upgrades. Furthermore, MontiTrans offers a comprehensive and generic solution that was developed not only for a specific DSL but also for any DSL. This makes the framework much more complex and heavyweight than necessary at this point. By tailoring a solution for the transformation from CD4A to GUIDSL, a lot of overhead could be saved and a lightweight easy-to-maintain tool could be created. Finally, MontiTrans is developed to transform models within a given DSL (endogenous transformation). Although MontiCore could enable MontiTrans to transform from a Model from one DSL to another, this would entail major code updates, and create a multitude of new challenges that need to be tackled in order to provide a generic solution.

10.6 Technology Readyness of MaCoCo

In this thesis, we claim that it is possible to create a full-size real-world system with this approach. For our evaluation, we will take a closer look at TRLs for MaCoCo. According to the ESA guidelines on TRLs for software development, [G+13, Com14] MaCoCo has reached the Technology Readiness Level 9:

TRL 1: MATHEMATICAL FORMULATION *Expression of a problem and of a concept of solution.*
In the project planning phase of MaCoCo a set of problems were defined that MaCoCo should address, such as: Financial management, staff management, and project management (*cf.* Section 8.1).

TRL 2: ALGORITHM *Practical application identified.*
A model-driven approach (*cf.* Chapter 7) was proposed in order to tackle the problems identified in TRL 1.

TRL 3: PROTOTYPE *Main use cases implemented.* An initial prototype was developed and tested with a group of lead users (cf. Section 8.2).

TRL 4: ALPHA version *Clear identification of the domain of applicability.*
In cooperation with the control committee, MaCoCo was adjusted in order to meet all requirements of the target domain, such as improved security and data privacy compliance.

TRL 5: BETA version *All use cases and error handling specified.*
MaCoCo has robust error handling, logging and monitoring. All initially specified use cases have been implemented.

TRL 6: Product RELEASE *All use cases and error handling implemented. User-friendliness validated.*
In cooperation with lead users the user-friendliness of MaCoCo is constantly validated and improved.

TRL 7: Early adopter version *Validity of solution confirmed within intended application. Requirements specification validated by the users.*
Lead-users accessed MaCoCo on a daily basis and confirmed its operation on a day-to-day basis. MaCoCo proved that it can be used by early adopters for its intended purpose as an information system and a controlling tool.

TRL 8: General product *Validity of solution confirmed within the intended application. Requirements specification validated by the users.* Feedback by early adopters was used to refine and complete the product.

TRL 9: Live product *Full process implemented, Maintenance, updates, etc.*
MaCoCo is released and published[14] [GHK+20, ANV+18]. A complete process is established governing error handling, maintenance, updates, and data migration, as well as user support, documentation, and developer onboarding. Note that as part of data privacy regulation and security, both documentation and the platform itself are only accessible from within the RWTH Aachen network: `macoco.rwth-aachen.de`.

Thus, MaCoCo meets all the requirements of a full-sized real-world application.

[14] `se-rwth.de/projects/MaCoCo`

Chapter 11

Conclusion

The following chapter summarizes the findings of this work and discusses the results of the respective elements of the toolchain.

Contents

11.1 Summary

This thesis makes contributions to the field of Model-Driven Development and web development, particularly in the context of integrating language model transformers. It offers new tools, methodologies, and practical applications, demonstrating the effectiveness of these innovations through real-world case studies and thorough theoretical exploration: We can break down the key contributions into the following areas:

Development of a Comprehensive Methodology: The thesis outlines a robust methodology focused on Model-Driven Development and web development. It emphasizes the integration of tools and practices that enhance usability, adaptability, variability, and real-world applicability.

Introduction of Key Transformers: A significant contribution is the development and implementation of three primary transformers: LLM4CD, CD2GUI, MontiGem. These transformers play a crucial role in the transformation process, from natural language to web application.

Development of New DSLs: One of the contributions is the development of new (GUIDSL v1, GUIDSL v2) specifically designed for the MontiGem transformer. This aspect signifies an advancement in the field, tailoring language models to specific needs in web development.

Validation of Transformers: The thesis validates the effectiveness of the LLM-based transformer (LLM4CD) through multiple tests, demonstrating its capability in transforming natural language into class diagrams—a significant achievement in bridging the gap between natural language processing and technical model development.

Functional Scope of Transformers: Detailed exploration of the functional scope, expansion stages, and limitations of the CD2GUI transformer is presented. The work also discusses the possibilities for extending and modifying both the transformers and the synthesized models, showing the adaptability and flexibility of the developed tools.

Real-World Application and Case Studies: A pivotal part of the work is the presentation of the MaCoCo case study, a real-world system used by RWTH Aachen University. This case study serves as a practical benchmark for validating the methodology. Additionally, the thesis explores various smaller projects, offering diverse applications and lessons learned.

Critical Discussion and Engagement with Related Work: The thesis includes a comprehensive discussion chapter where related work is examined. This not only contextualizes the research within the broader field but also provides critical insights and comparative analysis.

The thesis presented its contributions as follows: In Chapter 1 we introduce the goals and general motivation for the thesis. In addition, requirements for Usability, Adaptability, Variability and Transition to Real-World Systems are discussed. Next Chapter 2 presents the methodology providing an overview of the toolchain and pointing out the three key transformers used: LLM4CD, CD2GUI, MontiGem. In addition, the corresponding thesis structure is presented. Chapter 3 provides foundations of model-driven development and web development as well as introduces the DSLs re-used in this approach. In Chapter 4 we introduce the first transformer LLM4CD, and evaluate the capabilities of LLMs to produce models. In the following chapter (Chapter 5), the first transformer is presented as well as new DSLs developed for MontiGem. In order to validate the LLM-based transformer, multiple test results on the transformation from natural language to class diagrams are discussed. Once the first transformer and its target-DSL are presented, the next transformer can be discussed in Chapter 6. The functional scope of CD2GUI is presented together with expansion stages and limiting options of the transformer. In addition, options to extend and modify both the transformer as well as the synthesized models are shown. Next, the third transformer MontiGem is presented in Chapter 7. As MontiGem transforms models into a running application, this chapter presents both the run-time environment of the target application as well as the generated target code. Having iterated over all transformers of the methodology we present the first case study: MaCoCo. Chapter 8 provides insight into a full-size real-world system, that is currently in use by RWTH Aachen University at the time of writing. The system serves as a benchmark to validate the capabilities of the MontiGem-based methodology. As this approach was also used in further smaller projects, a selection of different use cases and the corresponding lessons learned is presented in Chapter 9. The chapter is followed by Chapter 10, in which the related work is discussed. The thesis is closed by this final Chapter 11.

11.2 Results

The goal of this thesis was the definition and evaluation of a *model-driven method for the development of a full-size web-based information system*. In developing a concept for an information system generator, we identified three challenges: (1) development of a model-driven tool that can produce such an application. (2) a tool that can interpret natural language input and transform it into a formal model and (3) a tool that transforms the output from one tool to the input the other one needs. Hence we defined three corresponding research questions (*cf.* Figure 11.1).

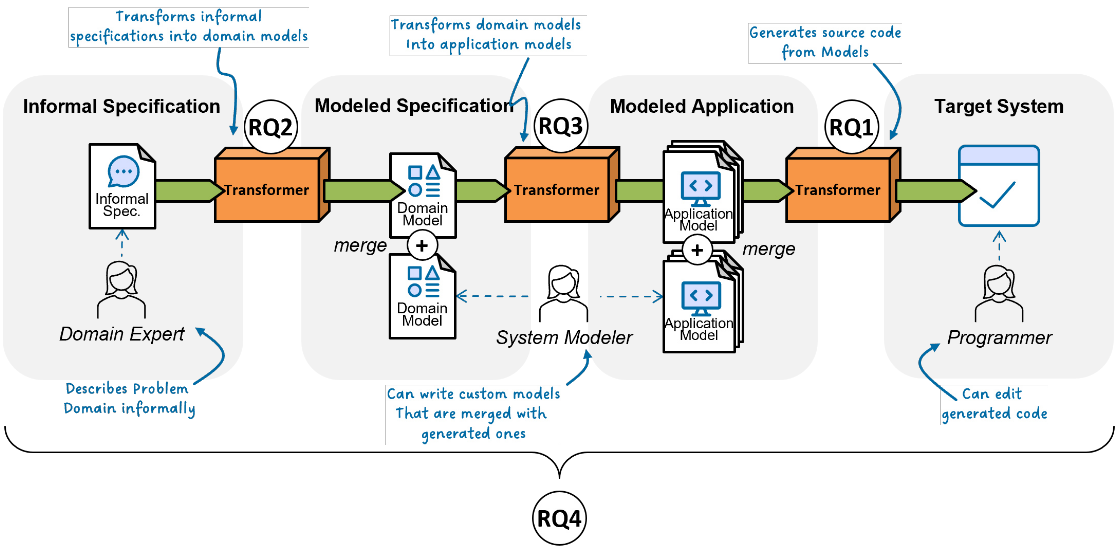


Figure 11.1: Toolchain Overview: Transforming Natural Language into a model, into models that define an applicaiton and finally into a target system

11.2.1 Model-Driven Development for Information Systems

The first research question introduced in Section 1.2 is defined as follows:

Research Question I
What is the structure of a model-driven approach that can be used to build a full-sized information system?

We propose a generator architecture in Chapter 7 that is capable of generating an information system. The architecture has been validated in multiple use cases presented in Chapter 8 and Chapter 9.

RQ1.1 What kind of models are needed to define an information system? We have shown that a prototype of a web-based information system can be defined using informal specifications, by transforming them into a domain-defining class diagram and deriving further models from this single underlying model. However, as we also have shown in Chapter 8, a real-world system such as MaCoCo requires a very precise and intricate definition, that can not be provided by a class diagram alone. Thus as the transformation of informal specifications is a very useful tool to rapidly produce a prototype, additional configurations must be made at least in the form of GUI-models (GUIDSL) and further class diagrams (CD4A) in order to leave the prototype phase of the application.
There are several approaches to defining an information system. In Section 10.5 we present model-driven approaches by Roth [Rot17] and Reiß [Rei16] both using data-centric methodologies to define their systems and define their underlying data structure using CD4A models, and define processes using activity diagrams (JavaAD). As we follow the Schneidermann Manta [CC05] (*cf.* Section 6.2) we use a combination of GUI-models and class diagrams to define the basis of our system. In addition, the approach allows for additional models such as tagging with OCL constraints in order to refine the system and add some validation logic to it. Thus we define our information system based on a set of CD4A and GUIDSL models, with the option to extend with additional models.

RQ1.2 How to transform a set of models into a running application? Transforming a set of models into a web information system is a hard challenge. Chapter 7 introduces the generator that was developed for this task. MontiGem follows the generator architecture proposed in [HKR21], and uses a set of M2M-transformations and template-based M2T-generation to produce domain-specific code for both the server and the client. This code is complimented by a domain-independent handwritten runtime environment. The resulting code forms the web application as presented in Chapter 8 and [BGK+23a, ANV+18].

RQ1.3 How to incrementally transform the application from a prototype to a full-sized real-world system Adaptability was considered throughout the entire methodology, leaving the domain expert, the system modeler, and the developer with options to modify the artifacts that they are provided from previous steps in the generation process. The domain expert can iteratively use LLM4CD in order to receive and optimize the root class diagram. This class diagram can also be merged with further models in order to ensure the target system contains specific data structures. Next the system modeler can extend and modify both class diagrams and GUI-models (cf. Section 6.6) in addition the templates used in the transformer CD2GUI can be exchanged providing a more systematic approach to configure the GUI-model-output. Finally, the developer can use the TOP-Mechanism [Rum17] in order to modify generated code. As all mechanisms described above are optional and work iteratively, the toolchain presented in the methodology is capable of both creating a prototype of a web application as well as a full-size real-world system, as shown in Chapter 8.

11.2.2 Using Large Language Models for Model-Driven Development

Research Question II

To what degree can large language models be used to transform natural language into a domain model?

In Chapter 4 we present the transformer LLM4CD that tackles the challenge to interpret and transform natural language into a CD4A model.

RQ2.1 To what degree are LLMs capable of reliably transforming natural language to a model of a given DSL? We presented the transformer LLM4CD (*cf.* Chapter 4) to transform informal specifications into class diagrams that define the domain of the targeted data-centric application. The transformer relies on LLMs in order to process the natural language input and therefore can not guarantee a valid model output for any given input. However, as discussed in Chapter 4, LLM4CD can be configured to efficiently produce CD4A models, with a success rate of up to 99%.

RQ2.2 How to use LLMs in an iterative process to incrementally create and modify a domain model? Although the chance of producing a valid CD4A model is high, there is still a chance, that LLM4CD does not produce a model that meets the requirement of the user. In this case, the transformer can be used iteratively. Produced models can be reused as input and modified using the LLM. As showcased in Chapter 4, external CD4A-models can be used as input and be adapted by LLM4CD.

11.2.3 Transforming Domain Models to Application Models

> **Research Question III**
> *What methodology is followed to translate platform-independent domain models into platform-specific models for building an information system?*

Having tackled both the model-driven creation of a web-based information system and the creation of models based on natural language, we still need to provide a solution on how to transform the natural-language-based models into the models required as input for the model-driven creation for web-applications. We provide another transformer for this task: CD2GUI.

RQ3.1 How can we derive models defining user interfaces of a data-centric information system for efficient data access? Chapter 6 presents a transformer that derives user interfaces for a given data structure model. The corresponding GUIDSL models are based on the principles of Schneiderman [CC05]. The derived user interfaces allow for efficient access and modification of the persistent data in the system. As demonstrated in Chapter 9.

RQ3.2 How can we ensure that the M2M transformation integrates seamlessly into the agile-iterative development process and allows for the customization of the generated models? The presented methodology provides the domain expert with tooling to produce a model-driven application by, deriving a set of models from informal specifications. However, these specifications have to be changeable as well as the models. Thus we have to ensure that once generated models are changed by a system modeler, the domain expert still can change the provided specifications without breaking the system as long as both changes are not in conflict. Section 6.6 introduces a mechanism similar to the TOP mechanism that allows the system-modeler to extend and modify synthesized models with handwritten ones. Thus any changes defined by the system modeler, will not be undone once CD2GUI will synthesize GUI-models in another iteration (*cf.* Section 6.7).

> **Research Question IV**
> *How to ensure adaptability, variability, and usability throughout a generative toolchain for information systems?*

The toolchain we presented in this thesis covers all three aspects of this research question.

RQ4.1 How to ensure adaptability of generated models and target code? Section 6.7 tackles the challenges of adaptability within a model-driven toolchain, and presents how they are addressed within this toolchain. CD2GUI implements mechanisms to replace templates depending on the provided single underlying model. Hence making the toolchain adapt more flexible to its input, rather than defining one generic user interface that has to match any use case.

RQ4.2 How to ensure variability of generated models and target code? Section 6.6 presents the elements in this approach that ensure variability, especially in models generated for the user interfaces. The models generated in this toolchain can be modified and extended. Especially GUI-models might require additional fine-tuning to meet the needs of the end user. Hence, models generated by CD2GUI can be modified by a system modeler. Changes are not lost when the application is regenerated.

RQ4.3 What UI-elements are required to ensure the usability of the generated information system? Section 6.2 presents the minimal set of pages and UI-Elements that are needed to have a reasonable amount of usability for the generated information system. The toolchain implements these features presented in these chapters an thus fulfills the requirements set up by research question IV. The toolchain relies on publications on user interface definitions, but also uses input from application of the system in the real world (*cf.* Chapter 8, Chapter 9) to maximize the usability of the initial prototype and to ensure the high usability of emerging real-world applications.

11.2.4 Technology Readyness Level of Produced Web applications

The toolchain is intended to produce a prototype that is iteratively transitioned to full-size software. In the following, we will take a closer look at both a produced prototype and a finished application.

Evaluating an Unconfigured Prototype

In order to evaluate a MontiGem-based web-application prototype a model-driven approach was implemented for the digitization of consulting documentation processes. We analyze the process of developing and deploying the application and its usability and performance. We compared the model-driven approach to a non-low-code software development approach. The evaluation of the resulting applications was based on ISO 25010 attributes, which include functional suitability, compatibility, security, maintainability, and user experience. MontiGem was compared to a conventional software development process that included common commercial frameworks such as Tailwind.

Usability As shown in Section 9.5 the usability of an unconfigured prototype is limited. The initial GUI components might not match the targeted domain and the navigation through the system might not be optimized to any processes a user might encounter. However, as we could show in Chapter 8, through iterative extensions, we can increase usability and produce a running real-world application that is used by a wide group of users.

Performance The experiments presented in Section 9.5 that handwritten solutions can be more responsive. However, we could show that the generated applications were performant enough for all generated use cases, and we could show that performance can be improved over time using our approach (*cf.* Figure 8.17). Handwritten code has the advantage that it is individually customized to increase the performance of the application. A generated prototype has not yet been optimized for this. We allow for optimization in all generated code an thus have the potential to be as performant as an entirely handwritten application.

Model-Driven development of Web Applications The developer tasked with using both approaches to implement the application describes the following limitations of the model-driven approach:

1. Limited functionality of the GUIDSL v1 compared to HTML and CSS, particularly in the area of input elements. The GUIDSL v1 has a limited set of input element types, which may not be sufficient for certain applications. Note that the development of GUIDSL v2 explicitly addressed this drawback (*cf.* Section 5.2).

2. Difficulty in customizing and modifying the GUIDSL v1. Making changes to the GUIDSL v1 or generator requires modifications to the underlying DSL and may be out of the scope of small projects. Note that the development of GUIDSL v2 explicitly addressed this drawback (*cf.* Section 5.2).

3. Lack of predefined GUI components in the GUIDSL v1 that are specific to the domain of the application. This may require finding alternative solutions or creating custom components.

4. The domain-specific focus of MontiGem, the model-driven framework used in the user study, resulted in design decisions that may not align with the requirements of the application. Certain features, such as a public registration functionality and input element types, were not readily available or required changes to or extensions of the framework.

5. The limitations of the GUIDSL and model-driven approach were more apparent in smaller projects with a single target platform. The benefits of the model-driven

approach may be more evident in larger projects that can afford changes to the DSLs and custom generators.

6. The potential for technology limitations when using a framework like MontiGem. Implementing certain features, such as offline functionalities or video streaming, may require different architectures in the front and backend, which may not align with the technologies utilized by the framework.

Many of the mentioned limitations address a limited number of available components and the limited flexibility of the modeling languages used. As mentioned in Section 5.2, the updated version of the DSL is designed to support component libraries and easy adaptation of used components providing both more flexibility and a greater selection of components. However, the MontiGem-based approach remains limited to the generation of web-based information systems.

Technological Readiness of the generated Application

We could show in Section 10.6 and Section 8.3 that the produced application can be transformed into an application that is used in production (TRL 9).

Generator evaluation (MontiGem)

MontiGem provides the developer with many artifacts for both the front end and back end, that define a well-structured and very systematic code base. Since this code is structured very systematically, it is easy to extend and adapt it. By using the generator, the development process becomes a little bit more complicated, as we have an extra process step, but with the increasing size of the project, the time cost of this step is negligible compared to the benefit of the automatically produced code. Projects such as MaCoCo and ADD have demonstrated the huge benefit of this approach [ANV+18]. Therefore, the extension should not affect the performance of the existing approach and should only offer further options and additional functions. Consequently, extensions comply with the following:

- **Optionality:** The extension should be optional and not force the developer to specify further models or configurations if he does not want to use this component.
- **Simple Build Process:** The extension should be easily integrated in the existing build process and add as little extra build steps as possible.
- **Compatability:** The extension mechanisms the developer is used to, such as the TOP-Mechanism should still be possible even with the target code produced by the plugin.

- **Ease of use and Documentation:** The barrier to entry for the developer should be kept as low as possible. Good documentation and 'getting started' material is provided.

Following a similar argument as in Section 10.6, we can place the generator at TRL 6.

11.2.5 Limitations

Although the presented methodology is very capable, we have to consider the following limitations: (1) As the first transformer relies on LLMs, we also inherit limitations based on this technology. LLM4CD configures and trains an LLM to produce a specific model, however, based on the input such training can also be undone. Giving instructions (such as 'Tell me the result of 4 + 4') instead of specifications as input will most likely result in an invalid model. This will be detected by the validation step in the process and the generator will abort. (2) LLMs have a chance to produce invalid results. Although we can configure LLM4CD to increase the chance of producing a valid model, we can not guarantee it. As neural nets such as LLMs produce results in a very time-efficient manner, LLM4CD can simply attempt again without any drawbacks to the methodology as a whole. (3) Handling obligatory associations is an unsolved issue of CD2GUI. Although we can produce user interfaces to create new objects, CD2GUI does not produce user interfaces that enable the user to create a new object while already creating one, as it would be needed for an obligatory bidirectional association. By extension CD2GUI also does not support the creation of objects that have circular obligatory associations, as all objects have to be created at the same time to produce a valid database state. Unidirectional associations can be handled by creating both objects in the correct order.

Bibliography

[50101] *Software Product Lines: Practices and Patterns.* Addison-Wesley Longman Publishing Co., Inc., USA, 2001.

[ADB13] José Luis Herrero Agustin and Pablo Carmona Del Barco. A model-driven approach to develop high performance web applications. *Journal of Systems and Software*, 86(12):3013–3023, 2013.

[AKJP05] Mohammed Arif, Dennis Kulonda, Jim Jones, and Michael Proctor. Enterprise information systems: technology first or process first? *Business Process Management Journal*, 2005.

[AMH21] Esra A Abdelnabi, Abdelsalam M Maatuk, and Mohammed Hagal. Generating uml class diagram from natural language requirements: A survey of approaches and techniques. In *2021 IEEE 1st International Maghreb Meeting of the Conference on Sciences and Techniques of Automatic Control and Computer Engineering MI-STA*, pages 288–293. IEEE, 2021.

[AMN+20] Kai Adam, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Enterprise Information Systems in Academia and Practice: Lessons learned from a MBSE Project. In *40 Years EMISA: Digital Ecosystems of the Future: Methodology, Techniques and Applications (EMISA'19)*, volume P-304 of *LNI*, pages 59–66. Gesellschaft für Informatik e.V., May 2020.

[And12] Chris Anderson. The model-view-viewmodel (mvvm) design pattern. In *Pro Business Applications with Silverlight 5*, pages 461–499. Springer, 2012.

[ANV+18] Kai Adam, Lukas Netz, Simon Varga, Judith Michael, Bernhard Rumpe, Patricia Heuser, and Peter Letmathe. Model-Based Generation of Enterprise Information Systems. In Michael Fellmann and Kurt Sandkuhl, editors, *Enterprise Modeling and Information Systems Architectures (EMISA'18)*, volume 2097 of *CEUR Workshop Proceedings*, pages 75–79. CEUR-WS.org, May 2018.

[Arn23] Arnaud Roques. PlantUML, 2023.

[BAGB14] Ankica Barišic, Vasco Amaral, Miguel Goulão, and Bruno Barroca. Evaluating the usability of domain-specific languages. In *Software Design and Development: Concepts, Methodologies, Tools, and Applications*, pages 2120–2141. IGI Global, 2014.

[BCC+22] Arvid Butting, Niel Conradie, Jutta Croll, Manuel Fehler, Clemens Gruber, Dominik Herrmann, Alexander Mertens, Judith Michael, Verena Nitsch, Saskia Nagel, Sebastian Pütz, Bernhard Rumpe, Elisabeth Schauermann, Johannes Schöning, Carolin Stellmacher, and Sabine Theis. Souveräne digitalrechtliche Entscheidungsfindung hinsichtlich der Datenpreisgabe bei der Nutzung von Wearables. In *Selbstbestimmung, Privatheit und Datenschutz : Gestaltungsoptionen für einen europäischen Weg*, pages 489–508. Springer Fachmedien Wiesbaden, April 2022.

[BCD+03] Mariano Belaunde, Cory Casanave, Desmond DSouza, Keith Duddy, William El Kaim, Alan Kennedy, William Frank, David Frankel, Randall Hauch, Stan Hendryx, et al. Mda guide version 1.0. 1, 2003.

[BDC23] Dominik Bork and Giuliano De Carlo. An extended taxonomy of advanced information visualization and interaction in conceptual modeling. *Data & Knowledge Engineering*, 147:102209, 2023.

[BDLD11] Mario Luca Bernardi, Giuseppe Antonio Di Lucca, and Damiano Distante. A model-driven approach for the fast prototyping of web applications. In *2011 13th IEEE International Symposium on Web Systems Evolution (WSE)*, pages 65–74. IEEE, 2011.

[BDM+24] Nils Baumann, Juan Sebastian Diaz, Judith Michael, Lukas Netz, Haron Nqiri, Jan Reimer, and Bernhard Rumpe. Combining retrieval-augmented generation and few-shot learning for model synthesis of uncommon dsls. In *Modellierung 2024 Satellite Events*, pages 10–18420. Gesellschaft für Informatik eV, 2024.

[BGK+23a] Constantin Buschhaus, Arkadii Gerasimov, Jörg Christian Kirchhof, Judith Michael, Lukas Netz, Bernhard Rumpe, and Sebastian Stüber. Lessons Learned from Applying Model-Driven Engineering in 5 Domains: The Success Story of the MontiGem Generator Framework. *Science of Computer Programming*, 232:103033, October 2023.

[BGK+23b] Constantin Buschhaus, Arkadii Gerasimov, Jörg Christian Kirchhof, Judith Michael, Lukas Netz, Bernhard Rumpe, and Sebastian Stüber. Lessons learned from applying model-driven engineering in 5 domains: The success story of the montigem generator framework. *Science of Computer Programming*, page 103033, 2023.

[BGRW17] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. Taming the Complexity of Model-Driven Systems Engineering Projects. In *Part of the Grand Challenges in Modeling (GRAND'17) Workshop*, July 2017.

[BGS05] Sven Burmester, Holger Giese, and Wilhelm Schäfer. Model-driven architecture for hard real-time systems: From platform independent models to code. In *Model Driven Architecture–Foundations and Applications: First European Conference, ECMDA-FA 2005, Nuremberg, Germany, November 7-10, 2005. Proceedings 1*, pages 25–40. Springer, 2005.

[BJP22] Shraddha Barke, Michael B James, and Nadia Polikarpova. Grounded copilot: How programmers interact with code-generating models. *arXiv preprint arXiv:2206.15000*, 2022.

[BK16] J Scott Brennen and Daniel Kreiss. Digitalization. *The international encyclopedia of communication theory and philosophy*, pages 1–11, 2016.

[BKRW17] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Architectural Programming with MontiArcAutomaton. In *In 12th International Conference on Software Engineering Advances (ICSEA 2017)*, pages 213–218. IARIA XPS Press, May 2017.

[BMR+20] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[BP20] K Backhaus and T Paulsen. Vom homo oeconomicus zum homo digitalis. *Marketing Weiterdenken: Zukunftspfade für eine marktorientierte Unternehmensführung*, 2:323–339, 2020.

[BRW16] Vincent Bertram, Bernhard Rumpe, and Michael von Wenckstern. Encapsulation, Operator Overloading, and Error Class Mechanisms in OCL. In *International Workshop in OCL and Textual Modeling (OCL'16)*, pages 17–32. ACM/IEEE, 2016.

[CBCR15] Tony Clark, Mark van den Brand, Benoit Combemale, and Bernhard Rumpe. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*, LNCS 9400, pages 7–20. Springer, 2015.

[CBT24] Javier Cámara, Lola Burgueño, and Javier Troya. Towards standarized benchmarks of llms in software modeling tasks: a conceptual framework. *Software and Systems Modeling*, pages 1–10, 2024.

[CC05] Brock Craft and Paul Cairns. Beyond guidelines: what can we learn from the visual information seeking mantra? In *Ninth International Conference on Information Visualisation (IV'05)*, pages 110–118. IEEE, 2005.

[CCOTF09] Maria Manuela Cruz-Cunha, Eva F Oliveira, Antonio J Tavares, and Luis G Ferreira. *Handbook of research on social dimensions of semantic technologies and web services*. IGI Global, 2009.

[CDDM09] Sven Casteleyn, Florian Daniel, Peter Dolog, and Maristella Matera. *Engineering web applications*, volume 30. Springer, 2009.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., USA, 2000.

[CFJ+16] Benoit Combemale, Robert France, Jean-Marc Jézéquel, Bernhard Rumpe, James Steel, and Didier Vojtisek. *Engineering Modeling Languages: Turning Domain Knowledge into Tools*. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series, November 2016.

[CGR23] Benoit Combemale, Jeff Gray, and Bernhard Rumpe. Large language models as an "operating" system for software and systems modeling. *Journal Software and Systems Modeling (SoSyM)*, 22(5):1091–1092, September 2023.

[Cle05] Paul Clements. *Software product lines practices and patterns*. The SEI series in software engineering. Addison-Wesley, 4. print. edition, 2005.

[CLS17] Xinyun Chen, Chang Liu, and Dawn Song. Towards synthesizing complex programs from input-output examples. *arXiv preprint arXiv:1706.01284*, 2017.

[CMNR24] Joel Charles, Judith Michael, Lukas Netz, and Bernhard Rumpe. Teaching model-driven low-code development platforms. 2024.

[Com14] European Commission. Technology readiness levels (trl). horizon 2020 work programme 2015. commission decision c (2014) 4995, 2014.

[CTBV23] Javier Cámara, Javier Troya, Lola Burgueño, and Antonio Vallecillo. On the assessment of generative ai in modeling tasks: an experience report with chatgpt and uml. *Software and Systems Modeling*, pages 1–13, 2023.

[CTJ+23] Yiannis Charalambous, Norbert Tihanyi, Ridhi Jain, Youcheng Sun, Mohamed Amine Ferrag, and Lucas C Cordeiro. A new era in software security: Towards self-healing software via large language models and formal verification. *arXiv preprint arXiv:2305.14752*, 2023.

[CYC+23] Kua Chen, Yujing Yang, Boqi Chen, Jose Hernandez Lopez, Gunter Mussbacher, and Daniel Varro. Automated domain modeling with large language models: A comparative study. In *26th International Conference on Model Driven Engineering Languages and Systems*, MODELS'23, 2023.

[Cza02] Krzysztof Czarnecki. Generative programming: Methods, techniques, and applications tutorial abstract. In *International Conference on Software Reuse*, pages 351–352. Springer, 2002.

[DAKM20] Bidit L Dey, Wafi Al-Karaghouli, and Syed Sardar Muhammad. Adoption, adaptation, use and impact of information systems during pandemic time and beyond: Research and managerial implications. *Information Systems Management*, 37(4):298–302, 2020.

[DBH18] Filip Karlo Došilović, Mario Brčić, and Nikica Hlupić. Explainable artificial intelligence: A survey. In *2018 41st International convention on information and communication technology, electronics and microelectronics (MIPRO)*, pages 0210–0215. IEEE, 2018.

[DBK23] Istvan David, Dominik Bork, and Gerti Kappel. Circular systems engineering. *arXiv preprint arXiv:2306.17808*, 2023.

[DGM+21] Imke Drave, Akradii Gerasimov, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. A Methodology for Retrofitting Generative Aspects in Existing Applications. *Journal of Object Technology (JOT)*, 20:1–24, November 2021.

[DHM+22] Manuela Dalibor, Malte Heithoff, Judith Michael, Lukas Netz, Jérôme Pfeiffer, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Generating Customized Low-Code Development Platforms for Digital Twins. *Journal of Computer Languages (COLA)*, 70, June 2022.

[DJR22] Florian Drux, Nico Jansen, and Bernhard Rumpe. A Catalog of Design Patterns for Compositional Language Engineering. *Journal of Object Technology (JOT)*, 21(4):4:1–13, October 2022.

[DRKdL+22] Davide Di Ruscio, Dimitris Kolovos, Juan de Lara, Alfonso Pierantonio, Massimo Tisi, and Manuel Wimmer. Low-code development and model-driven engineering: Two sides of the same coin? *Software and Systems Modeling*, 21(2):437–446, 2022.

[DRRS09] Michael Dukaczewski, Dirk Reiss, Bernhard Rumpe, and Mark Stein. MontiWeb - Modular Development of Web Information Systems. In *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM' 09)*, Orlando, Florida, USA, October 2009.

[DS02] Paulo Pinheiro Da Silva. *Object modelling of interactive systems: the UMLi approach*. The University of Manchester (United Kingdom), 2002.

[Eck95] Wayne W Eckerson. Three tier client/server architecture: Achieving scalability, performance and efficiency in client server applications. *Open Information Systems*, 10(1), 1995.

[Elm90] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[ENT15] Meryem Elallaoui, Khalid Nafil, and Raja Touahni. Automatic generation of uml sequence diagrams from user stories in scrum process. In *2015 10th international conference on intelligent systems: theories and applications (SITA)*, pages 1–6. IEEE, 2015.

[ET12] David W Embley and Bernhard Thalheim. *Handbook of conceptual modeling: theory, practice, and research challenges*. Springer, 2012.

[FADB+22] James Finnie-Ansley, Paul Denny, Brett A Becker, Andrew Luxton-Reilly, and James Prather. The robots are coming: Exploring the implications of openai codex on introductory programming. In *Australasian Computing Education Conference*, pages 10–19, 2022.

[FCK+95] David Ferraiolo, Janet Cugini, D Richard Kuhn, et al. Role-based access control (rbac): Features and motivations. In *Proceedings of 11th annual computer security application conference*, pages 241–48, 1995.

[Fel23] Eric Feltgen. Analysis of a model-driven approach to the digitization of consulting documentation processes, 2023.

[FFK23] Hans-Georg Fill, Peter Fettke, and Julius Köpke. Conceptual modeling and large language models: impressions from first experiments with chatgpt. *Enterprise Modelling and Information Systems Architectures (EMISAJ)*, 18:1–15, 2023.

[FGH+23] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. Large language models for software engineering: Survey and open problems. *arXiv preprint arXiv:2310.03533*, 2023.

[FL10] Martin Fowler and Domain-Specific Languages. Addison-wesley professional, 2010.

[Fle79] Rudolf Flesch. How to write plain english. *University of Canterbury. Available at http://www. mang. canterbury. ac. nz/writing_guide/writing/flesch. shtml.[Retrieved 5 February 2016]*, 1979.

[Fou22] Apache Software Foundation. Apache tomee, 2022.

[Fow10] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.

[FP00] Piero Fraternali and Paolo Paolini. Model-driven development of web applications: the autoweb system. *ACM Transactions on Information Systems (TOIS)*, 18(4):323–382, 2000.

[FR07] Robert France and Bernhard Rumpe. Model-driven Development of Complex Software: A Research Roadmap. *Future of Software Engineering (FOSE '07)*, pages 37–54, May 2007.

[G+13] ESA TRL Working Group et al. Guidelines for the use of trls in esa programmes. Technical report, Technical Report ESSB-HB-E-002, European Space Agency, 2013.

[GHJ+95] Erich Gamma, Richard Helm, Ralph Johnson, Ralph E Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[GHK+20] Arkadii Gerasimov, Patricia Heuser, Holger Ketteniß, Peter Letmathe, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Generated Enterprise Information Systems: MDSE for Maintainable Co-Development of Frontend and Backend. In Judith Michael and Dominik Bork, editors, *Companion Proceedings of Modellierung 2020 Short, Workshop and Tools & Demo Papers*, pages 22–30. CEUR Workshop Proceedings, February 2020.

[GHL+22] Arkadii Gerasimov, Patricia Heuser, Peter Letmathe, Judith Michael, Lukas Netz, Bernhard Rumpe, Simon Varga, and Galina Volkova. Domain Modelling of Financial, Project and Staff Management, April 2022. Zenodo, https://doi.org/10.5281/zenodo.6422355.

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0: Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008.

[GLRR15] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. Engineering Tagging Languages for DSLs. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 34–43. ACM/IEEE, 2015.

[GM01] Athula Ginige and San Murugesan. The essence of web engineering-managing the diversity and complexity of web application development. *IEEE multimedia*, 8(2):22–25, 2001.

[GMN+20] Arkadii Gerasimov, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Continuous Transition from Model-Driven Prototype to Full-Size Real-World Enterprise Information Systems. In Bonnie Anderson, Jason Thatcher, and Rayman Meservy, editors, *25th Americas Conference on Information Systems (AMCIS 2020)*, AIS Electronic Library (AISeL), pages 1–10. Association for Information Systems (AIS), August 2020.

[GMNR21] Arkadii Gerasimov, Judith Michael, Lukas Netz, and Bernhard Rumpe. Agile Generator-Based GUI Modeling for Information Systems. In Ajantha Dahanayake, Oscar Pastor, and Bernhard Thalheim, editors, *Modelling to Program (M2P)*, pages 113–126. Springer, March 2021.

[GR18] Jeff Gray and Bernhard Rumpe. Agile model-based system development. *Journal Software and Systems Modeling (SoSyM)*, 17(4):1053–1054, 2018.

[Grö10] Hans Grönniger. Systemmodell-basierte Definition objektbasierter Modellierungssprachen mit semantischen Variationspunkten. In *Aachener Informatik-Berichte, Software Engineering Band 4*. Shaker Verlag, 2010.

[GWT+13] Matthias Galster, Danny Weyns, Dan Tofan, Bartosz Michalik, and Paris Avgeriou. Variability in software systems—a systematic literature review. *IEEE Transactions on Software Engineering*, 40(3):282–306, 2013.

[GWT+14] Matthias Galster, Danny Weyns, Dan Tofan, Bartosz Michalik, and Paris Avgeriou. Variability in software systems—a systematic literature review. *IEEE Transactions on Software Engineering*, 40(3):282–306, 2014.

[H+06] Matthew Hause et al. The sysml modelling language. In *Fifteenth European Systems Engineering Conference*, volume 9, pages 1–12, 2006.

[HBAA18] Maryum Hamdani, Wasi Haider Butt, Muhammad Waseem Anwar, and Farooque Azam. A systematic literature review on interaction flow modeling language (ifml). In *Proceedings of the 2018 2nd International Conference on Management Engineering, Software Engineering and Service Sciences*, pages 134–138, 2018.

[HH13] John Hunt and John Hunt. Gang of four design patterns. *Scala Design Patterns: Patterns for Practical Reuse and Design*, pages 135–136, 2013.

[HHK+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Software Product Line Conference (SPLC'13)*, pages 22–31. ACM, 2013.

[HJ16] Steven Hirshorn and Sharon Jefferies. Final report of the nasa technology readiness assessment (tra) study team. Technical report, 2016.

[HJK+23] Malte Heithoff, Nico Jansen, Jörg Christian Kirchhof, Judith Michael, Florian Rademacher, and Bernhard Rumpe. Deriving Integrated Multi-Viewpoint Modeling Languages from Heterogeneous Modeling Languages: An Experience Report. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2023, page 194â€“207, Cascais, Portugal, October 2023. Association for Computing Machinery.

[HKR21] Katrin Hölldobler, Oliver Kautz, and Bernhard Rumpe. *MontiCore Language Workbench and Library Handbook: Edition 2021.* Aachener Informatik-Berichte, Software Engineering, Band 48. Shaker Verlag, May 2021.

[HLMSN+15a] A. Haber, M. Look, P. Mir Seyed Nazari, A. Navarro Perez, B. Rumpe, S. Völkel, and A. Wortmann. *Composition of Heterogeneous Modeling Languages.* Springer International Publishing, 2015.

[HLMSN+15b] A. Haber, M. Look, P. Mir Seyed Nazari, A. Navarro Perez, S. Völkel, and A. Wortmann. Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In *3rd International Conference on Model-Driven Engineering and Software Development*. SciTePress, 2015.

[HM02] Jan Heering and Marjan Mernik. Domain-specific languages for software engineering. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences*, pages 3649–3650. IEEE, 2002.

[HMPO+08] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Gøran K. Olsen, and Andreas Svendsen. Adding standardized variability to domain specific languages. In *2008 12th International Software Product Line Conference*, pages 139–148, 2008.

[Ho16] J Hammond and Kony outSystems. The forrester wave™: Mobile low-code development platforms, q1 2017. *Forrester Research, Cambridge*, 2016.

[Höl18] Katrin Hölldobler. *MontiTrans: Agile, modellgetriebene Entwicklung von und mit domänenspezifischen, kompositionalen Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 36. Shaker Verlag, December 2018.

[HR17] Katrin Hölldobler and Bernhard Rumpe. *MontiCore 5 Language Workbench Edition 2017*. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017.

[HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208. Springer, 2012.

[HRW15] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 136–145. ACM/IEEE, 2015.

[HŠB21] Tjaša Heričko, Boštjan Šumak, and Saša Brdnik. Towards representative web performance measurements with google lighthouse. In *Proceedings of the 2021 7th Student Computer Science Research Conference*, page 39, 2021.

[HSVT21] Johannes Hintsch, Daniel Staegemann, Matthias Volk, and Klaus Turowski. Low-code development platform usage: towards bringing citizen development and enterprise it into harmony. 2021.

[HWDB20] Thomas L Huber, Maike AE Winkler, Jens Dibbern, and Carol V Brown. The use of prototypes to bridge knowledge boundaries in agile software development. *Information systems journal*, 30(2):270–294, 2020.

[Hyk23] Solomon Hykes. Docker, 2023.

[HZL+23] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *arXiv preprint arXiv:2308.10620*, 2023.

[Jac97] Ivar Jacobson. *Software reuse architecture process and organization for business success*. Addison-Wesley-Longman, Harlow, England, first printed edition, 1997.

[JHC21] Matthew D Jones, Scott Hutcheson, and Jorge D Camba. Past, present, and future barriers to digital transformation in manufacturing: A review. *Journal of Manufacturing Systems*, 60:936–948, 2021.

[JP21] Samkit Jain and Pradnya Purandare. Study of the usability testing of e-commerce applications. In *Journal of Physics: Conference Series*, volume 1964, page 042059. IOP Publishing, 2021.

[JR23] Nico Jansen and Bernhard Rumpe. Seamless Code Generator Synchronization in the Composition of Heterogeneous Modeling Languages. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2023, page 163â€“168, Cascais, Portugal, October 2023. Association for Computing Machinery.

[KH17] Ratnakar Kumar and Nitasha Hasteer. Evaluating usability of a web application: A comparative analysis of open-source tools. In *2017 2nd International Conference on Communication and Electronics Systems (ICCES)*, pages 350–354. IEEE, 2017.

[Kin21] Gavin King. Hibernate, 2021.

[KJN19] Achaporn Kwangsawad, Aungkana Jattamart, and Paingruthai Nusawat. The performance evaluation of a website using automated evaluation tools. In *2019 4th Technology Innovation Management and Engineering Science International Conference (TIMES-iCON)*, pages 1–5. IEEE, 2019.

[KR18] Monique F Kilkenny and Kerin M Robinson. Data quality:“garbage in–garbage out”, 2018.

[Kro23] Lukas Kronast. Introducing variability into model-driven user interface development, 2023.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe'08)*, LNBIP 11, pages 297–315. Springer, 2008.

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

[KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain-specific modeling: enabling full code generation*. John Wiley & Sons, 2008.

[LCV22] Márk Lajkó, Viktor Csuvik, and László Vidács. Towards javascript program repair with generative pre-trained transformer (gpt-2). In *Proceedings of the Third International Workshop on Automated Program Repair*, pages 61–68, 2022.

[Lid11] Stephen W Liddle. Model-driven software development. In *Handbook of Conceptual Modeling: Theory, Practice, and Research Challenges*, pages 17–54. Springer, 2011.

[LMPV96] Lila F Laux, Peter R McNally, Michael G Paciello, and Gregg C Vanderheiden. Designing the world wide web for people with disabilities: a user centered design approach. In *Proceedings of the second annual ACM conference on Assistive technologies*, pages 94–101, 1996.

[Lo23] David Lo. Trustworthy and synergistic artificial intelligence for software engineering: Vision and roadmaps. *arXiv preprint arXiv:2309.04142*, 2023.

[Loo17a] M. Look. *Unterstützung modellgetriebener, agiler Entwicklung mehrbenutzerfähiger, ubiquitärer Enterprise Applikationen durch Generatoren.* PhD thesis, RWTH Aachen University, Aachen, 2017.

[Loo17b] Markus Look. *Modellgetriebene, agile Entwicklung und Evolution mehrbenutzerfähiger Enterprise Applikationen mit MontiEE.* Aachener Informatik-Berichte, Software Engineering, Band 27. Shaker Verlag, March 2017.

[LPKW06] Henry Lieberman, Fabio Paternò, Markus Klann, and Volker Wulf. End-user development: An emerging paradigm. In *End user development*, pages 1–8. Springer, 2006.

[LRSS23] Achim Lindt, Bernhard Rumpe, Max Stachon, and Sebastian Stüber. Cdmerge: Semantically sound merging of class diagrams for software component integration. *Journal of Object Technology*, 22(2):2:1–14, July 2023.

[LVM+04] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, Murielle Florins, and Daniela Trevisan. Usixml: A user interface description language for context-sensitive user interfaces. In *Proceedings of the ACM AVI 2004 Workshop*, 2004.

[Mar82] James Martin. *Application development without programmers.* Prentice Hall PTR, 1982.

[Mar91] James Martin. *Rapid application development.* Macmillan Publishing Co., Inc., 1991.

[MBLK23] Thomas McGill, Oluwaseun Bamgboye, Xiaodong Liu, and Chathuranga Sampath Kalutharage. Towards improving accessibility of web auditing with google lighthouse. In *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1594–1599. IEEE, 2023.

[MCF03] Stephen J Mellor, Anthony N Clark, and Takao Futagami. Guest editors' introduction: Model-driven development. *IEEE Software*, 20(05):14–18, 2003.

[MdS15] Francisco Morais and Alberto Rodrigues da Silva. Assessing the quality of user-interface modeling languages. In *International Conference on Enterprise Information Systems*, volume 2, pages 311–319. SCITEPRESS, 2015.

[MEBF+19] Zoë Meleo-Erwin, Corey Basch, Joseph Fera, Danna Ethan, and Philip Garcia. Readability of online patient-based information on bariatric surgery. *Health promotion perspectives*, 9(2):156, 2019.

[MFBF+22] Silverio Martínez-Fernández, Justus Bogner, Xavier Franch, Marc Oriol, Julien Siebert, Adam Trendowicz, Anna Maria Vollmer, and Stefan Wagner. Software engineering for ai-based systems: a survey. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(2):1–59, 2022.

[MIT23] Ariana Martino, Michael Iannelli, and Coleen Truong. Knowledge injection to counter large language model (llm) hallucination. In *European Semantic Web Conference*, pages 182–185. Springer, 2023.

[MLG23] Potsawee Manakul, Adian Liusie, and Mark JF Gales. Selfcheckgpt: Zero-resource black-box hallucination detection for generative large language models. *arXiv preprint arXiv:2303.08896*, 2023.

[MNN+22] Judith Michael, Imke Nachmann, Lukas Netz, Bernhard Rumpe, and Sebastian Stüber. Generating Digital Twin Cockpits for Parameter Management in the Engineering of Wind Turbines. In *Modellierung 2022*, pages 33–48. Gesellschaft für Informatik, June 2022.

[MNRV19] Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Towards Privacy-Preserving IoT Systems Using Model Driven Engineering. In Nicolas Ferry, Antonio Cicchetti, Federico Ciccozzi, Arnor Solberg, Manuel Wimmer, and Andreas Wortmann, editors, *Proceedings of MODELS 2019. Workshop MDE4IoT*, pages 595–614. CEUR Workshop Proceedings, September 2019.

[MSN17] P. Mir Seyed Nayari. *MontiCore: Efficient Development of Composed Modeling Language Essentials.* PhD thesis, RWTH Aachen University, Aachen, 2017.

[MSNRR15a] P. Mir Seyed Nazari, A. Roth, and B. Rumpe. Mixed Generative and Handcoded Development of Adaptable Data-centric Business Applications. In *Proceedings of the Workshop on Domain-Specific Modeling.* ACM, 2015.

[MSNRR15b] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. Mixed Generative and Handcoded Development of Adaptable Data-centric Business Applications. In *Domain-Specific Modeling Workshop (DSM'15)*, pages 43–44. ACM, 2015.

[mtpG20] mgm technology partners GmbH. Low code und co-innovation fuer individuelle enterprise software, 2020.

[mtpG21] mgm technology partners GmbH. Widget showcase, 2021.

[MWLN22] Dang Minh, H Xiang Wang, Y Fen Li, and Tan N Nguyen. Explainable artificial intelligence: a comprehensive review. *Artificial Intelligence Review*, pages 1–66, 2022.

[NGM+24] Lukas Netz, Arkadii Gerasimov, Judith Michael, Bernhard Rumpe, and Peter Letmathe. Modeling financial, project and staff management: A case report from the macoco project. *Enterprise Modelling and Information Systems Architectures (EMISAJ)*, 19, 2024.

[NK20] Nguyen Nguyen and Taehong Kim. Toward highly scalable load balancing in kubernetes clusters. *IEEE Communications Magazine*, 58(7):78–83, 2020.

[NMR24a] Lukas Netz, Judith Michael, and Bernhard Rumpe. From natural language to web applications: Using large language models for model-driven software engineering. In *Modellierung 2024*, pages 179–195. Gesellschaft für Informatik e.V., Bonn, 2024.

[NMR24b] Lukas Netz, Judith Michael, and Bernhard Rumpe. From natural language to web applications: Using large language models for model-driven software engineering. In *Modellierung 2024*, pages 179–195. Gesellschaft für Informatik eV, 2024.

[NPAB22] Olivia D. Negoita, Mirona A. Popescu, Albena Antonova, and Dominik Bork. A comparison of business platforms used by smes to digitalize management activities, 2022. Name - European Commission; European Union; Copyright - © 2022. This work is published under http://archive.ceciis.foi.hr/app/index.php/ceciis/archive (the "License"). Notwithstanding the ProQuest Terms and Conditions, you may use this content in accordance with the terms of the License; Zuletzt aktualisiert - 2022-10-04.

[Obj17] Object Management Group. OMG Unified Modeling Language (OMG UML), 2017.

[OKH+22] Jaehoon Oh, Sungnyun Kim, Namgyu Ho, Jin-Hwa Kim, Hwanjun Song, and Se-Young Yun. Understanding cross-domain few-shot learning based on domain similarity and few-shot difficulty. In *Advances in Neural Information Processing Systems*, 2022.

[Ope23] OpenAI. Gpt-4 technical report, 2023.

[Ort22] Kai Ortmanns. Generating data structure based forms for data entry in web information systems, 2022.

[PAA+23] Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, Kranthi Kiran GV, et al. Rwkv: Reinventing rnns for the transformer era. *arXiv preprint arXiv:2305.13048*, 2023.

[PRH21] Niculin Prinz, Christopher Rentrop, and Melanie Huber. Low-code development platforms-a literature review. In *AMCIS*, 2021.

[RA17] J Rymer and Kony Appian. The forrester wave™: Low-code development platforms for ad&d pros, q4 2017. *Cambridge, MA: Forrester Research*, 2017.

[RdS14] André Ribeiro and Alberto Rodrigues da Silva. Xis-mobile: A dsl for mobile applications. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 1316–1323, 2014.

[Rei16] Dirk Reiß. *Modellgetriebene generative Entwicklung von Web-Informationssystemen.* Aachener Informatik-Berichte, Software Engineering, Band 22. Shaker Verlag, May 2016.

[Ren15] David Rensin. *Kubernetes.* O'Reilly Media, Incorporated, 2015.

[RJW03] Melanie Ruhe, Ross Jeffery, and Isabella Wieczorek. Cost estimation for web applications. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 285–294. IEEE, 2003.

[RKA12] Faisal Rahutomo, Teruaki Kitasuka, and Masayoshi Aritsugi. Semantic cosine similarity. In *The 7th international student conference on advanced science and technology ICAST*, volume 4, page 1, 2012.

[RKL+19] John R Rymer, Rob Koplowitz, Salesforce Are Leaders, Kony Mendix, Salesforce are Leaders, GeneXus ServiceNow, Strong Performers, WaveMaker MatsSoft, and Thinkwise are Contenders. The forrester wave™: Low-code development platforms for ad&d professionals, q1 2019. *Forrester Report, Forrester*, 2019.

[Rot17] Alexander Roth. *Adaptable Code Generation of Consistent and Customizable Data Centric Applications with MontiDex.* Aachener Informatik-Berichte, Software Engineering, Band 31. Shaker Verlag, December 2017.

[RR13] Dirk Reiss and Bernhard Rumpe. Using Lightweight Activity Diagrams for Modeling and Generation of Web Information Systems. In *Proceedings 4th International United Information Systems Conference, UNISCON 2012*, volume 137 of *Lecture Notes in Business Information Processing*, pages 61–73, Yalta, Ukraine, June 2013. Springer.

[RRM+14] Clay Richardson, JR Rymer, Christopher Mines, Alex Cullen, and Dominique Whittaker. New development platforms emerge for customerfacing applications (2014). *URL: https://www. forrester. com/report/New+ Development*, 52, 2014.

[RRW13] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring, 2te Auflage*. Springer Berlin, Juni 2012.

[Rum17] Bernhard Rumpe. *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International, May 2017.

[SBLY08] JinGang Shi, YuBin Bao, FangLing Leng, and Ge Yu. Study on log-based change data capture and handling mechanism in real-time data warehouse. In *2008 international conference on computer science and software engineering*, volume 4, pages 478–481. IEEE, 2008.

[SC01] Nary Subramanian and Lawrence Chung. Software architecture adaptability: an nfr approach. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 52–61, 2001.

[Sch97] Ken Schwaber. Scrum development process. In Jeff Sutherland, Cory Casanave, Joaquin Miller, Philip Patel, and Glenn Hollowell, editors, *Business Object Design and Implementation*, pages 117–134, London, 1997. Springer London.

[Sch20] Henri Schildt. *The data imperative: How digitalization is reshaping management, organizing, and work*. Oxford University Press, USA, 2020.

[SCS23] Mohammed Latif Siddiq, Beatrice Casey, and Joanna Santos. A lightweight framework for high-quality code generation. *arXiv preprint arXiv:2307.08220*, 2023.

[Sel03] Bran Selic. The pragmatics of model-driven development. *IEEE software*, 20(5):19–25, 2003.

[SFK+00] Ravi Sandhu, David Ferraiolo, Richard Kuhn, et al. The nist model for role-based access control: towards a unified standard. In *ACM workshop on Role-based access control*, volume 10, 2000.

[Shn96] B. Shneiderman. The eyes have it: a task by data type taxonomy for information visualizations. In *Proceedings 1996 IEEE Symposium on Visual Languages*, pages 336–343, 1996.

[Sla22] Ivan Slavov. Generating navigation variants for enterprise information system prototypes, 2022.

[Som11] Ian Sommerville. Software engineering (ed.). *America: Pearson Education Inc*, 2011.

[SS94] Ravi S Sandhu and Pierangela Samarati. Access control: principle and practice. *IEEE communications magazine*, 32(9):40–48, 1994.

[ST19] Klaus-Dieter Schewe and Bernhard Thalheim. *Design and development of web information systems*. Springer, 2019.

[SVD21] Richard Shin and Benjamin Van Durme. Few-shot semantic parsing with language models trained on code. *arXiv preprint arXiv:2112.08696*, 2021.

[TLL+23] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. Is chatgpt the ultimate programming assistant–how far is it? *arXiv preprint arXiv:2304.11938*, 2023.

[TMN+04] Hideki Tai, Kinichi Mitsui, Takashi Nerome, Mari Abe, Kouichi Ono, and Masahiro Hori. Model-driven development of large-scale web applications. *IBM Journal of Research and Development*, 48(5.6):797–809, 2004.

[TMR+23] Nicholas D Thomas, Raegan Mahler, Matthew Rohde, Nicole Segovia, and Kevin G Shea. Evaluating the readability and quality of online patient education materials for pediatric acl tears. *Journal of Pediatric Orthopaedics*, 43(9):549–554, 2023.

[TWT22] Jakob Trischler and Jessica Westman Trischler. Design for experience–a public service design approach in the age of digitalization. *Public Management Review*, 24(8):1251–1270, 2022.

[USMDAS14] A Ubale Swapnaja, G Modani Dattatray, and S Apte Sulabha. Analysis of dac mac rbac access control based models for security. *International Journal of Computer Applications*, 104(5):6–13, 2014.

[VAT+20] Alvaro Veizaga, Mauricio Alferez, Damiano Torre, Mehrdad Sabetzadeh, Lionel Briand, and Elene Pitskhelauri. Leveraging natural-language requirements for deriving better acceptance criteria from models. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS '20, page 218–228, New York, NY, USA, 2020. Association for Computing Machinery.

[VGBS01] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. On the notion of variability in software product lines. In *Proceedings Working IEEE/IFIP Conference on Software Architecture*, pages 45–54. IEEE, 2001.

[VID+19] Paul Vincent, Kimihiko Iijima, Mark Driver, Jason Wong, and Yefim Natis. Magic quadrant for enterprise low-code application platforms. *Gartner report*, 2019.

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.

[VZG22] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*, pages 1–7, 2022.

[WBEJ20] Diana White, River Bond, Joshua Eastes, and Negar Janani. Representing and interpreting data from playfair. 2020.

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012.

[WK+23] Tongshuang Wu, Kenneth Koedinger, et al. Is ai the better programming partner? human-human pair programming vs. human-ai pair programming. *arXiv preprint arXiv:2306.05153*, 2023.

[WTB+22] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*, 2022.

[WZK+23] Martin Weyssow, Xin Zhou, Kisub Kim, David Lo, and Houari Sahraoui. On the usage of continual learning for out-of-distribution generalization in pre-trained language models of code. *arXiv preprint arXiv:2305.04106*, 2023.

[XL05] Weidong Xia and Gwanhoo Lee. Complexity of information systems development projects: conceptualization and measurement development. *Journal of management information systems*, 22(1):45–83, 2005.

[XX13] Spyros Xanthopoulos and Stelios Xinogalos. A comparative analysis of cross-platform development approaches for mobile applications. In *Pro-*

ceedings of the 6th Balkan Conference in Informatics, pages 213–220, 2013.

[YS22] Song Yang and Houari Sahraoui. Towards automatically extracting uml class diagrams from natural language specifications. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 396–403, 2022.

[YYH23] Hui Yang, Sifu Yue, and Yunzhong He. Auto-gpt for online decision making: Benchmarks and additional opinions. *arXiv preprint arXiv:2306.02224*, 2023.

[ZKX+23] Xin Zhou, Kisub Kim, Bowen Xu, Jiakun Liu, DongGyun Han, and David Lo. The devil is in the tails: How long-tailed code distributions impact large language models. *arXiv preprint arXiv:2309.03567*, 2023.

[Zur18] Berne Lausanne Lugano Zurich. Software engineering and architectures. *Master of Science in Engineering*, page 424, 2018.

[ZWZ+20] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. Retrieval-based neural source code summarization. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1385–1397, 2020.

Appendix A

Code Listings

The following chapter contains selected code listings that are intended to aid in the understanding of this thesis.
The following table lists the repositories of the primary projects involved in this thesis:

Artefact	Chapter	Repository
MontiCore	Chapter 3	`https://github.com/MontiCore/monticore`
CD4A	Chapter 3	`https://git.rwth-aachen.de/monticore/cd4analysis/cd4analysis`
Tagging	Chapter 3	`https://git.rwth-aachen.de/monticore/montigem/cdtagging`
LLM4CD	Chapter 4	`https://git.rwth-aachen.de/se/ai4se`
GUIDSL	Section 5.1	`https://git.rwth-aachen.de/monticore/languages/gui-dsl`
CD2GUI	Chapter 6	`https://git.rwth-aachen.de/monticore/montigem/cd2gui`
MaCoCo	Chapter 8	`https://git.rwth-aachen.de/macoco/implementation`
MontiGem	Chapter 7	`https://git.rwth-aachen.de/monticore/montigem/montigem-gen`
UMLP	Section 9.6	`https://git.rwth-aachen.de/monticore/umlp`

A.1 Domain Models

The following CD4A-Models are domain models serving as the single underlying model for web applications.
The MaCoCo class diagram:

```
/* (c) https://github.com/MontiCore/monticore */
/* adapted and customized by (c) MaCoCo, ein RWTH Aachen projekt */

package de.macoco.be.domain;

import java.time.ZonedDateTime;

classdiagram MaCoCo {

  class Person {
    String vorname;
    Optional<String> vorsatzwort;
```

```
    String nachname;
    String kuerzel;       // Abgeleitet analog wie bei SAP: 3 Vor- 3 Nachnamenbuchstaben
    Optional<String> personalnummer;
    Optional<ZonedDateTime> gebDatum;
    Optional<ZonedDateTime> beschBeginn;
    Optional<ZonedDateTime> beschEnde;
    List<String> kommentar;
    boolean istAktiv;
    boolean istStundenzettelpflichtig = false;
    Optional<ZonedDateTime> stundenzettelpflichtigVon;
    Optional<ZonedDateTime> stundenzettelpflichtigBis;
    Optional<Long> ueberStundenUebertrag;
    boolean darfWochenendarbeiten;

    Optional<ZonedDateTime> hoechstbeschaeftigungBis;
    List<String> staatsangehoerigkeiten;
    Optional<String> telefonnummer;
    Optional<ZonedDateTime> arbeitserlaubnisBis;
    boolean extern;
    Optional<String> titel;
    Optional<String> rufname;
    Optional<String> adresse;
    Optional<String> email;
    Optional<String> geburtsname;
    Optional<String> geschlecht;
    Optional<String> lBVNummer;
    Optional<ZonedDateTime> entfristung;
    <<dbColumnDefinition="TEXT">>
    Optional<String> qualifikationstitel;
    boolean speicherungGewuenscht = false;
    <<dbColumnDefinition="TEXT">>
    Optional<String> urlaubskommentar;
    Optional<ZonedDateTime> aenderungsdatum;
  }

  association [1] Person -> (anstellungsarten) Anstellungsart [*];
  association [*] Person -> (vorgesetzte) Person [*];
  association [1] Person -> (notiz) Freitext [0..1];

  association [1] Person -> (jahresurlaubstage) Jahresurlaub [*];
  association [1] Person -> (zusatzurlaubstage) Sonderurlaub [*];
  association [1] Person -> (stundenzettel) Stundenzettel [*];
  association [1] Person -> (abwesenheiten) Abwesenheit [*];

    <<nocascade>>
  association [1] Person -> (user) MacocoUser [0..1];
```

```
class Anstellungsart {
  BeschaeftigungsArt beschArt;
}

association [1] Anstellungsart -> (vertraege) Vertrag [*];
association [1] Anstellungsart -> (anstellungsformen) Anstellungsform
    [*];

enum BeschaeftigungsArt {
  WiMi("Wissenschaftliche/r Mitarbeiter/in (WiMi)"),
  HiWi("Hilfskraft (HiWi)"),
  BTV("Beschaeftigte/r in Technik und Verwaltung (BTV)"),
  BEAMTE("Beamte/r"),
  AZUBI("Azubi/ne"),
  PLAN;
}


class Vertrag {
  ZonedDateTime vertragsBeginn;
  ZonedDateTime vertragsEnde;
  String vertragsStatus; // Aktiv, Planung, aenderungsvertrag,
      Abgelaufen
  List<String> kommentar;
  ZahlenWert planUmfang;
  Optional<Long> arbeitsstundenProWoche; //Stundenzettel
  boolean kuendigungsschutz;
  Optional<String> vertragsgrundlage;
  Optional<String> aktion;
  Optional<ZonedDateTime> aktionsDatum;
}

association [1] Vertrag -> (kostenstellen) Kostenstelle [*];

class Anstellungsform {
  boolean erstanstellung;
  Optional<String> entgeltGruppe; // { TVL-13}  Fuer Hiwis: {SHK (
      default), WHB, WHK}
  Optional<String> erfahrungsStufe; // 1- 6 ; sowie Spezialfaelle (TVL
      ue+);
  boolean unbefristet;
  ZonedDateTime anstellungVon;
  Optional<ZonedDateTime> anstellungBis;
  long gehaltCent;
  boolean istEigenerReferenzwert;
  List<String> kommentar;
  Optional<String> berufsgruppe;
  Optional<String> aktion;
  Optional<ZonedDateTime> aktionsDatum;
}
```

```
class Kostenstelle {
  ZonedDateTime verbuchungsBeginn;
  ZonedDateTime verbuchungsEnde;
  KostenstelleBezeichnung bezeichnung;
  List<String> kommentar;
  ZahlenWert beschaeftigungsUmfang;
  boolean buchungenNeuErzeugen;
  boolean buchungenLoeschen;
  boolean gesperrt;
}

enum KostenstelleBezeichnung {
  NONE(""),
  AN_INSTITUT("An-Institut"),
  GMBH("GmbH"),
  ANDERER_LEHRSTUHL,
  ANDERES_INSTITUT,
  VORUEBERGEHEND_ABWESEND("Voruebergehend Abwesend"),
  ANDERE_FINANZIERUNG;
}

enum Zuweisungsart {
  KONTO("Kontenzuweisung"),
  STELLENZUWEISUNG,
  PLANSTELLE,
  SONSTIGES;
}

association [*] Kostenstelle -> (personalBudget) Budget [0..1];
association Kostenstelle -> (stellenzuweisung) Stellenzuweisung [0..1];
association Kostenstelle -> (planstelle) Planstelle [0..1];
                            <<nocascade>>
association [0..1] Kostenstelle -> (personalBuchungen) Buchung [*];

class Planstelle {
  String bezeichnung;
  Optional<String> minEntgeltgruppe;
  Optional<String> minEntgeltstufe;
  Optional<String> maxEntgeltgruppe;
  Optional<String> maxEntgeltstufe;
  Optional<ZonedDateTime> verfuegbarVon;
  Optional<ZonedDateTime> verfuegbarBis;
  Optional<String> kommentar;
  ZahlenWert planUmfang;
  boolean aktiv;
}

association Planstelle -> (konto) Konto [0..1];
```

```
class ZahlenWert {
  ZahlenTyp zahlenTyp;
  long wert;
}

enum ZahlenTyp {EURO, STUNDE, PROZENT, NONE; }

abstract class Konto {
  String name;
  Optional<String> pspElement;
  Optional<String> kontotyp;  // unterschiedlich je nach Kontoart
  Optional<ZonedDateTime> sapDatum;
  Optional<String> internesAktenzeichen;
  boolean istPlanKonto;
  boolean istVerbuchungsKonto;
  boolean istAktiv;
  Optional<String> vergabeVerordnung;
  Optional<String> farbe;
  Optional<ZonedDateTime> aenderungsdatum;
  /Optional<ZonedDateTime> optStartDatum;
  /Optional<ZonedDateTime> optEndDatum;
  /Geschaeftsvorgang gueltigerGeschaeftsvorgang;
}

<<treatAsBidirectional>>
association [1] Konto (konto) -> (gesamtBudget) Budget [0..1];
association [1] Konto -> (kommentare) Freitext [*];
association [1] Konto -> (notiz) Freitext [0..1];
association [1] Konto <-> MailAlert [*];
association [1] Konto -> (abgleichsKonto) ExternKonto [0..1];

abstract class ErweitertesKonto extends Konto {
  Optional<ZonedDateTime> startDatum;
  Optional<ZonedDateTime> endDatum;
  Optional<ZonedDateTime> bewilligungsDatum;    // Bewilligungsdatum
      ist im Industrieprojekt nur mandatory, wenn kein Sammelkonto
  Optional<ZonedDateTime> verlaengertBisDatum;
  Optional<ZonedDateTime> aufstockungsDatum;
  Optional<Long> foerderquote;
  Optional<String> aktenzeichen;
  Optional<Long> finanzierteKomplettsumme;
  Optional<String> referenzSponsor;
  boolean hatProgrammpauschale;
}

class Drittmittelprojekt extends ErweitertesKonto {
}

association [*] Drittmittelprojekt -> (fachlicherVerantwortlicher)
    Person [*];
```

```
class Haushaltskonto extends Konto {
  Optional<ZonedDateTime> startDatum;
  Optional<ZonedDateTime> endDatum;
}

class Industrieprojekt extends ErweitertesKonto {
  Optional<String> auftraggeber;
  boolean istSammelkonto;
  Optional<Long> mehrwertsteuer;
}

association [*] Industrieprojekt -> (fachlicherVerantwortlicher) Person
     [*];

class Sonstiges extends ErweitertesKonto {
  Optional<String> auftraggeber;
  Geschaeftsvorgang geschaeftsvorgang;
  Optional<Long> mehrwertsteuer;
  boolean istSammelkonto;
}

enum AbrechnungsInterval {
  GESAMTE_PROJEKTLAUFZEIT("Gesamte Projektlaufzeit"),
  JAEHRLICH("Jaehrlich"),
  HALBJAEHRLICH("Halbjaehrlich"),
  QUARTALSWEISE("Quartalsweise"),
  MONATLICH("Monatlich");
}

association [1] Konto -> OrganigrammKonto [*];

class OrganigrammKonto {
  Optional<ZonedDateTime> von;
  Optional<ZonedDateTime> bis;
  Optional<ZahlenWert> umfang;
}

association [*] OrganigrammKonto -> Organigramm [1];

class ZeitlicheVerbuchungskontoZuordnung {
  Optional<Long> programmPauschale;
  Optional<Long> gemeinkostensatz;
  Optional<ZonedDateTime> startDate;
  Optional<ZonedDateTime> endDate;
  boolean istHauptVerbuchungskonto;
}

association [1] ErweitertesKonto -> (verbuchungskontoZuordnung)
    ZeitlicheVerbuchungskontoZuordnung [*];
```

```
association [*] ZeitlicheVerbuchungskontoZuordnung -> (verbuchungskonto
    ) Konto [0..1];

class Budget {
  String typ; // name
  Optional<ZonedDateTime> startDatum;
  Optional<ZonedDateTime> endDatum;
  Optional<String> kommentar;
  Optional<long> budgetRahmenCent; // wird im Gesamtbudget als
      Bewilligungssumme interpretiert
  Optional<long> eigenAnteilCent;
  List<long> jahresBudgets;
  boolean proportionaleVerteilung;
  boolean forOverheads; // das erste Budget (gesamtBudget.
      getUnterbudgets[0])
  Optional<String> budgetKategorie;
  /int budgetDepth;
}

association [*] Budget -> Konto [1];
association [0..1] Budget (elternBudget) <-> (unterBudget) Budget [*];
association [1] Budget (budget) <-> (buchungseintrag) Buchungseintrag
    [*];
association [1] Budget (budget) <-> (stellenzuweisung) Stellenzuweisung
     [*];
<<noOrphanRemoval, noRemoveCascade>>
association [0..1] Budget -> Organigramm [0..1];

abstract class Buchungseintrag {
  ZonedDateTime datum;
  Optional<String> buchungseintragText; // human name: Buchung: "
      Buchungstext", Rechnung: "Rechnungsgrund", Mittelabruf: "Anmerkung
      ", Mittelzuweisung: "Grund"
  long betragCent;  // human name: "Betrag", Rechnung: "Rechnungsbetrag
      ", Konten mit Programmpauschale: "Rechnungsbetrag (inkl Pauschale)
      "
  boolean istAktiv;
  Optional<ZonedDateTime> aenderungsdatum;
  /Geschaeftsvorgang gueltigerGeschaeftsvorgang;
  /String lfdeNummer;
}

class Buchung extends Buchungseintrag {
  ZonedDateTime belegdatum;
  String zahlungsgrund; //Belegpositionstext
  Optional<String> kreditorDebitor;
  List<String> sachkonto;
  Optional<ZonedDateTime> buchungsdatum;
  BuchungsStatus status;
  List<String> belegnummern;
```

```
    List<String> auftragsnummer;
    Optional<String> bereich;
    Optional<String> projekt;
    Optional<ZonedDateTime> geschaeftsjahr;
    Optional<Long> steuer;
    Optional<Long> skonto;
    Optional<Long> gez_Skonto;
    Optional<String> erfasser;
    Optional<String> belegart;
    Optional<String> referenz;
    Optional<String> stornonummer;
    Optional<String> stkz;
    Optional<ZonedDateTime> erfassungsdatum;
    Optional<ZonedDateTime> ausgleichsdatum;
    long betragCentOriginal; // Bei Abweichungen von BetragCent kann nun
        eine Absicht ueberprueft werden
    boolean budgetChangedManually; // s.o.
    Optional<String> kostenarten;
    /Optional<Integer> bezugsdatumJahr;
  }

  class Rechnungsstellung extends Buchungseintrag {
    Optional<ZonedDateTime> rechnungsdatum;
    Optional<String> rechnungsnummer;
    RechnungsstellungStatus status;
    List<String> belegnummern;
    Optional<Long> gemeinkosten;
    Optional<Long> honorierung;
    Optional<String> projekt;
    Optional<String> debitor;
    Optional<Long> mehrwertsteuer;
    Optional<ZonedDateTime> zahlungsziel;
  }

  class Mittelzuweisung extends Buchungseintrag {
    Optional<ZonedDateTime> verfallDatum;
    Optional<String> kennung;
    MittelzuweisungStatus status;
  }

  class Mittelabruf extends Buchungseintrag {
    ZonedDateTime abrufdatum;
    MittelabrufStatus status;
    String zeitraum;
    Optional<Long> pauschaleManuell; // Praesent, falls die Pauschale
        manuell geaendert wurde
  }

  class Stellenzuweisung {
    ZonedDateTime erstellDatum;         // aktuelles Datum
```

```
    Optional<ZonedDateTime> startDatum; // Datum, zu welchem die Stelle beginnt
    Optional<ZonedDateTime> endDatum;   // Datum, zu welchem die Stelle endet
    long wert;
    String kennung;
    StellenzuweisungStatus status;
    ZahlenWert stellenumfang;
    boolean istAktiv;
    Optional<ZonedDateTime> aenderungsdatum;
  }

  enum Geschaeftsvorgang {
    NONE,
    MITTELABRUF,
    ZUWEISUNG,
    RECHNUNG;
  }

  enum BuchungsStatus {
    EINGEREICHT,
    SAP( "SAP"),
    SAP_STORNIERT( "SAP(Storniert)"),
    PLANUNG,
    FEHLERHAFT;
  }

  enum RechnungsstellungStatus {
    OFFENE_RECHNUNG,
    SAP( "SAP"),
     SAP_STORNIERT( "SAP(Storniert)"),
    PLANUNG;
  }

  enum MittelzuweisungStatus {
    BESCHEID_ERHALTEN ( "Bescheid erhalten"),
    SAP( "SAP"),
    SAP_STORNIERT( "SAP(Storniert)"),
    PLANUNG,
    FEHLERHAFT;
  }

  enum MittelabrufStatus {
    ABGERUFEN,
    SAP( "SAP"),
     SAP_STORNIERT( "SAP(Storniert)"),
    PLANUNG;
  }

  enum StellenzuweisungStatus {
```

```
    BESETZT,
    TEILBESETZT,
    UNBESETZT,
    PLANUNG;
  }

  enum VertragStatus {
    AKTIV,
    PLANUNG,
    AENDERUNGSVERTRAG("aenderungsvertrag"),
    ABGELAUFEN;
  }

  class SAPVerbindung {
    SAPverbindungsStatus status;
  }

  class Anfrage {
    String ikz;
    String bezeichner;
    boolean letzteAnfrageErfolgreich;
  }

  enum SAPverbindungsStatus {
    VERBUNDEN,
    GETRENNT,
    PROBLEM;
  }

  enum SAPimportMode {
    DEFAULT,
    F1KONTO;
  }

  association [1] SAPVerbindung -> (authorizedBy) MacocoUser [*];
  association [1] MacocoUser -> (anfragen) Anfrage [*];
  association [1] Anfrage -> (filteredKonten) PSPImportFilter [*];

  class AbgleichsBuchung extends Buchung {
      AbgleichsStatus abgleichsStatus;
      Optional<String> abgleichKommentar;
      String positionsNummer;
      boolean deactivated;
  }

    enum AbgleichsStatus {
      ABGESCHLOSSEN,            // Der Abgleich dieser Buchung ist Abgeschlossen
      TEILWEISE_ABGESCHLOSSEN,  // Der Abgleich wurde schon Bearbeitet es fehlen jedoch Summen
```

```
    ABWEICHEND,               // Der Abgleich ist Vollstaendig mit
        Abweichenden Summen
    OFFEN,                    // Der Abgleich wurde noch nicht gesetzt
    VORGESCHLAGEN;            // Der Abgleich wurde vom System
        vorgeschlagen
  }

class AbgleichsGruppe {
  String bezeichnung;
}
                              <<nocascade>>
association AbgleichsGruppe -> (buchungen) Buchung [*];
                              <<nocascade>>
association AbgleichsGruppe -> (abgleichsBuchungen) AbgleichsBuchung
    [*];
                               <<nocascade>>
association AbgleichsGruppe -> (konto) Konto [1];


class Freitext {
  Optional<ZonedDateTime> erstellDatum;
  Optional<ZonedDateTime> bearbeitetDatum;
                                  <<dbColumnDefinition="TEXT">>
  Optional<String> text;
}

abstract class Projekt {
  String name;
  String kuerzel;
  Optional<Long> minPM;
  Optional<Long> maxPM;
  Optional<Long> finanzPM;
  Optional<ZonedDateTime> laufzeitVon;
  Optional<ZonedDateTime> laufzeitBis;
  Optional<String> kommentar;
  Optional<String> foerdergeberAZ;
  ProjektStatus status;
  boolean istAktiv;
  boolean lockedForStundenzettel;   // Das gesamte Projekt ist gesperrt
  List<ZonedDateTime> lockedMonths; // Stellen die Ausnahme aus der
      Lock-Regelung fuer das gesamte Projekt dar
  Optional<ZonedDateTime> aenderungsdatum;
}

association [1] Projekt -> Aufwand [*];
association [1] Projekt -> Arbeitspaket [*];
association [1] Projekt -> Anstellung [*];
association [*] Projekt -> Konto [0..1];
association [*] Projekt -> (hauptverantwortlich) Person [*];
association [1] Projekt -> (notiz) Freitext [0..1];
```

```
class Auftragsprojekt extends Projekt {
  Optional<String> nummer;
  ProjektTyp typ;
  Optional<String> regelung;
}

class Organisation extends Projekt {
}

class Lehre extends Projekt {
  Optional<Long> stunden;
}

class Arbeitspaket {
  String nummer; // koennte auch 1.1 sein
  String name;
  Optional<String>  beschreibung;
  Optional<ZonedDateTime> beginnDatum;
  Optional<ZonedDateTime> endeDatum;
  Optional<Long> pMs;
  Optional<Long> stunden;
  Optional<ZonedDateTime> aenderungsdatum;
}

class Aufwand {
  /long pM;
  ZahlenWert umfang;
  ZonedDateTime laufzeitVon;
  ZonedDateTime laufzeitBis;
  Optional<ZonedDateTime> aenderungsdatum;
}

association [*] Aufwand -> Person [0..1];

class Anstellung {
  Optional<String> bezeichnung;
  ZahlenWert umfang;
  ZonedDateTime von;
  ZonedDateTime bis;
  boolean verfaellt;
  Optional<Stellentyp> min;
  Optional<Stellentyp> max;
  /long pM;
  BeschaeftigungsArt beschaeftigungsArt;
}

association [*] Anstellung -> Person [0..1];

class Stellentyp {
```

```
    Optional<String> entgeltgruppe;
    Optional<String> entgeltstufe;
  }

  enum ProjektStatus {
    IN_DEFINITION,
    BEANTRAGT,
    GENEHMIGT,
    ABGELEHNT,
    LAUFEND,
    FACHLICH_ABGESCHLOSSEN("Fachlich abgeschlossen"),
    ENDBERICHT_EINGEREICHT("Endbericht eingereicht"),
    INPRUEFUNG("In Pruefung"),
    GEPRUEFT("Geprueft"),
    ABGERECHNET,
    ABGESCHLOSSEN,
    ARCHIVIERT;
  }

  enum ProjektArt {
    AUFTRAG,
    ORGANISATION,
    LEHRE,
    NONE;
  }

  enum ProjektTyp {
    HOHEITLICH,
    INDUSTRIE,
    UNIVERSITAET("Universitaet"),
    SONSTIGE;
  }

  enum Vergabeverordnung {
    VOL_A ("VOL/A"),
    VOB_A ("VOB/A"),
    UVGO_NRW ("UVgO NRW"),
    UVGO_BUND ("UVgO Bund"),
    NONE (" ");
  }

  class Stundenzettel {
    StundenzettelStatus status;
    ZonedDateTime zeit;
    Optional<Long> abgegebenVonUserId;
    Optional<ZonedDateTime> abgabeDatum;
  }

  association [1] Stundenzettel -> (eintraege) StundenzettelEintrag [*];
```

```
class StundenzettelEintrag {
  ZonedDateTime uhrzeitVon;
  Optional<ZonedDateTime> uhrzeitBis;
  Optional<String> beschreibung;
  /long stunden;
  StundenzettelProjekt pauseOderSonstiges;
}

association [*] StundenzettelEintrag ->  Projekt [0..1];
association [*] StundenzettelEintrag ->  Arbeitspaket [0..1];

class StundenzettelHistorieEintrag {
  ZonedDateTime bearbeitungsdatum;
  Optional<Long> editorId;
  Optional<String> editedBy;
  String monat;
  int jahr;
  String changeOperation;
}

enum StundenzettelChangeOperation {
  INVALID,
  ABGEGEBEN("Abgegeben"),
  INTERN_GEPRUEFT("Intern Geprueft"),
  INTERN_ABGESCHLOSSEN("Intern Abgeschlossen");
}

association [1] Person ->  StundenzettelHistorieEintrag [*];

class Abwesenheit {
  Abwesenheitsgrund grund;
  ZonedDateTime  datumVon;
  ZonedDateTime datumBis;
  Optional<String> kommentar;
  Optional<ZonedDateTime> uhrzeitVon;
  Optional<ZonedDateTime> uhrzeitBis;
  Optional<Long> minutenProTag;
  /long tageGesamt;
}

class Dienstreise extends Abwesenheit {
  Optional<String> reisenummer;
  Optional<String> dienstreisegrund;
  Optional<String> reiseziel;
  Optional<Long> gesamtkosten;
  Optional<String> status;
  Optional<String> statusText;
}

association [*] Abwesenheit -> Konto [0..1];
```

```
association [1] Abwesenheit -> UrlaubsGenehmigung [0..1];

association [*] Dienstreise -> Budget [0..1];

class Arbeitstage {
  ZonedDateTime guiltigAb;
  ZonedDateTime guiltigBis;
  List<Integer> wochenTage; // 0..4 (Montag-Freitag)
}

association [1] Person -> Arbeitstage [*];

class Jahresurlaub {
  int jahr;
  long tageAnzahl;
  Optional<Long> stundenAnzahl;
  boolean istManuellerEintrag;
}

class UrlaubsGenehmigung {
  Optional<ZonedDateTime> beantragt;
  Optional<ZonedDateTime> genehmigt;
  Optional<ZonedDateTime> geprueft;
  Optional<ZonedDateTime> storniert;
  Urlaubsstatus status;
  Stornierungsstatus stornierungsStatus;
}

class Sonderurlaub {  // Fuer WZL: Zusatzurlaub umbenennen in
    Sonderurlaub und neues Feld anzahl
  ZonedDateTime  datumVon;
  ZonedDateTime datumBis;
  long anzahl;
}

class UrlaubEinstellungen {
  boolean urlaubsanspruchInStunden;
}

enum Abwesenheitsgrund {
  U_URLAUB,
  D_DIENSTREISE,
  S_SONSTIGE;
}

enum Urlaubssstatus {
  NONE,
  BEANTRAGT,
  GEPRUEFT("Geprueft"),
```

```
    PRUEFUNG_ABGELEHNT("Pruefung abgelehnt"),
    GENEHMIGT,
    GENEHMIGUNG_ABGELEHNT;
  }

  enum Stornierungsstatus {
    NONE,
    VERBOTEN,
    STORNIERUNGSANTRAG,
    STORNIERT;
  }

  enum Abrechnungsstatus {
    OFFEN,
    ABZURECHNEN,
    ABGERECHNET,
    STORNIERT;
  }

  enum StundenzettelStatus {
    IN_ERFASSUNG,
    INTERN_INPRUEFUNG("Intern in Pruefung"),
    INTERN_ABGESCHLOSSEN,
    DRITTMITTELABTEILUNG_INPRUEFUNG("Drittmittelabteilung in Pruefung"),
    DRITTMITTELABTEILUNG_ABGESCHLOSSEN,
    ENDBERICHT_EINGEREICHT,
    GEPRUEFT("Geprueft"),
    FOERDERGEBER_ABGESCHLOSSEN("Foerdergeber abgeschlossen"),
    ABGESCHLOSSEN;
  }

  enum StundenzettelProjekt {
    PAUSE,
    SONSTIGE,
    NONE;
  }

  association [*] F1Konto -> (institut) Institut [0..1];

  class ExternKonto extends Konto {
    Optional<ZonedDateTime> startDatum;
    Optional<ZonedDateTime> endDatum;
    Optional<String> importVermerk;
    AbgleichType abgleichVerhalten;
  }

  enum AbgleichType {
    IGNORE,
    MANUEL,
    AUTO_COPY;
```

```
}

association [*] ExternKonto -> (institut) Institut [0..1];

class PSPImportFilter{
  boolean importThisPSPelement;
  boolean importBookings;
  String name;
  String pspElement;
  Optional<ZonedDateTime> startDate;
}

association [*] PSPImportFilter -> (targetPSP) Konto [0..1];

//------------------------------------------------
// Reports

class EventReport {
  ZonedDateTime eventStart;
  Optional<ZonedDateTime> eventEnd;
  /EventStatus status;
  String message;
}

class EventReportEntry {
  EventType type;
  Optional<Long> targetId; // click -> navigate to id
  Optional<Integer> count;
  <<dbColumnDefinition="TEXT">>
  String message; // Zeile 4 Ergebniss ...... Zeile 5 .....
  EventStatus status;
}
association [1] EventReport -> (entries) EventReportEntry [*];

class ImportReport extends EventReport {
}

class ImportReportEntry extends EventReportEntry {
}


class PersonalExportReport extends EventReport {
  Long dataId;

  ZonedDateTime datenStart;
  ZonedDateTime datenEnd;
  Long countData;
  /Long countError;
  EventStatus exportStatus;
}
```

```
class PersonalExportReportEntry extends EventReportEntry {
  Optional<String> vorsatzwort;
  String vorname;
  String nachname;
  BeschaeftigungsArt beschaeftigungsArt;
  ZonedDateTime monat;
}

enum EventStatus {
  // Import
  SUCCESS("erfolgreich"),
  ERROR("fehlerhaft"),
  INFO("erfolgreich"), // success + special
  CREATED("erstellt"),
  UPDATED("upgedated"),
  IGNORED("ignoriert"),
  IN_PROGRESS("in bearbeitung"),

  // Personal Export
  SENT("verschickt"),
  CORRECT("sachlich korrekt");
}

enum EventType {
  IMPORT_KONTO("Konten"),
  IMPORT_BUCHUNG("Buchungs"),
  IMPORT_PERSON("Personen"),
  IMPORT_INSTITUT("Instituts"),
  IMPORT_NUTZER("Nutzer"),
  IMPORT_VERTRAG("Vertrags"),
  EXPORT_FINANZEN_KONTO("Konten"),
  EXPORT_FINANZEN_PLANSTELLE("Planstellen"),
  EXPORT_FINANZEN_VERTRAG("Vertragabdeckungs");
}

// Institut
<<dbColumn = "unique=true">>
class Institut {
  String institutsKennZiffer;
  String institutsName;
  String professorName;
  boolean lokalInstitut;
}

// Fachgruppe
enum Fachgruppe {
  MATHEMATIK,
  INFORMATIK,
  PHYSIK,
```

```
  CHEMIE,
  BIOLOGIE;
}

//SAP-Abschoepfung von Geschaeftsjahr
class GeschaeftsjahrSAPAbschoepfung {
  String fachgruppe;
  String institutsKennZiffer;
  String institutsName;
  String pspElement;
  long gesamtSAPAbschoepfung;
}

class SAPAbschoepfung {
  Long betragCent;
  ZonedDateTime year;
}

association [1] GeschaeftsjahrSAPAbschoepfung -> (sapAbschoepfung)
    SAPAbschoepfung [*];


// Konto
class F1Konto extends Haushaltskonto {
  Optional<ZonedDateTime> startDatum;
  Optional<ZonedDateTime> endDatum;
  long originalBudgetCent;
  long sonstigeZuweisungenCent;
  long resteCent;
  long aktuellerKontostandCent;
  long kontoRahmenCent;
  KommunikationsStatus kommunikationsStatus;
  StrafsteuerStatus strafsteuerStatus;
  long strafsteuerBasisCent;
  long strafsteuerSAPCent;
  long strafsteuerCent;
  long strafsteuerBerechnungsGrundlageCent;
}

association [1] Buchung -> (begruendung) Begruendung [0..1];

class Begruendung {
  String text;
  ZonedDateTime erstellung;
  Optional<ZonedDateTime> letzteBearbeitung;
  Optional<StrafsteuerGeschaeftsjahr> geschaeftsjahr;
}

association [1] Begruendung -> (akzeptanz) Akzeptanz [0..1];
```

```
class Akzeptanz {
  Akzeptanzstatus akzeptanzstatus;
  Optional<long> teilbetragCent;
  Optional<ZonedDateTime> bearbeitung;
  boolean istAktiv;
  Optional<String> kommentar;
}

enum KommunikationsStatus {
  KOM_STAT_NO_ACTION_NEEDED,        // Kein Handlungsbedarf
  KOM_STAT_BEGR_FEHLT,              // Begruendung fehlt
  KOM_STAT_ANTW_FEHLT,              // Antwort fehlt
  KOM_STAT_ANTW_CHANGED,            // Buchung oder Begruendung wurden
      geaendert
  KOM_STAT_BEGR_CHANGED,            // Antwort wurde geaendert
  KOM_STAT_ANTW_BEGR_CHANGED,       // Antwort und Begruendung wurde
      geaendert
  KOM_STAT_ANTW_FEHLT_CHANGED;      // Antwort Fehlt, andere Antwort
      wurde geaendert
}

enum StrafsteuerStatus {
  NONE,
  SOME,
  ALL;
}

enum Akzeptanzstatus {
  OFFEN,
  OK,
  TEILOK,
  NOTOK;
}

class StrafsteuerGeschaeftsjahr {
  ZonedDateTime jahr;
  boolean current;
}

class StrafsteuerSperrDatum {
  ZonedDateTime sperrDatum;
}

class MacocoUser {
                                    <<dbColumn = "unique=true">>
  String username;
  Optional<String> encodedPassword;
  String passwordSaltBase64;
  ZonedDateTime registrationDate;
  Optional<String> initials;
```

```
    MacocoUserActivationStatus activated;
    boolean enabled;
                                                                  <<dbColumn = "unique=true">>
    String email;
    boolean authentifiziert;
    Optional<String> timID;
    Optional<String> sapAccessToken;
    Optional<String> sapRefreshToKen;

  }

  association  [*] MacocoUser -> (institute) Institut [*];

  enum MacocoUserActivationStatus {
    AKTIVIERT,
    MAIL_NICHT_GESENDET("Mail nicht gesendet"),
    MAIL_FEHLERHAFT("Mail fehlerhaft"),
    MAIL_GESENDET("Mail gesendet");
  }

  // Favoriten
  class Favorite {
    String title;
    String url;
  }

  association [1] MacocoUser -> (favorite) Favorite [*];

  class RoleAssignment {
  }

  association [*] RoleAssignment -> (user) MacocoUser [1];
  association [*] RoleAssignment -> (accessPolicy) AccessPolicy [1];

  class AccessPolicy {
    String roleName;
    Optional<Long> objId;
  }

  class GroupedAccessPolicy {
    String name;
  }

  association [*] GroupedAccessPolicy -> (user) MacocoUser [*];
  association [1] GroupedAccessPolicy -> (policy) AccessPolicy [*];

  class Frist {
    String beschreibung;
    Friststatus status;
    Fristobjekt fristobjekt;
```

```
    ZonedDateTime faelligkeitsdatum;
    Optional<String> notiz;
    Boolean manuellerEintrag;
    Long bezugsobjektId;
    String bezugsobjektName;
    Optional<ZonedDateTime> aenderungsdatum;
  }

  association [*] Frist -> (zuweisung) Person [0..1];

  association [1] Person -> Frist [*];
  association [1] Konto -> Frist [*];
  association [1] Projekt -> Frist [*];
  association [1] Mittelabruf -> Frist [0..1];
  association [1] Rechnungsstellung -> Frist [0..1];
  association [1] Vertrag -> Frist [0..1];
  association [1] Zulage -> Frist [0..1];

  enum Friststatus {
    PLANUNG,
    IN_BEARBEITUNG,
    ERLEDIGT;
  }

  enum Fristobjekt {
    PERSON,
    KONTO,
    VERTRAGSVERLAENGERUNG("Vertragsverlaengerung"),
    ZULAGE,
    RECHNUNG,
    MITTELABRUF,
    PROJEKT;
  }

  class DefaultFristen {
    ZonedDateTime finanzenStartDatum;
    Long finanzenFristinWeeks;
    ZonedDateTime personalStartDatum;
    Long personalFristinWeeks;
  }

  class Setting {
    String identifier;
  }

  class InstanzSetting extends Setting {
    String value;
  }

  class UISetting extends Setting {
```

```
    String seite;
  }

  association [0..1] MacocoUser -> (setting) UISetting [*];

  class FilterSetting extends UISetting {
    List<String> filterValues;
  }

  class TableSetting extends UISetting {
    TableIdentifierTyp typ;
    boolean zeigeInaktive;
    int zeilenLimit;
    Optional<String> gruppiereNach;
    Optional<String> sortiereNach;
    Optional<String> sortiereDir;
    boolean isShowColor;
  }

  association [1] TableSetting -> (spalten) TableSettingSpalten [*];

  enum TableIdentifierTyp {
    DEFAULT,
    USER,
    INSTITUT;
  }

  class TableSettingSpalten {
    String name;
    boolean istAktiv;
    long breite;
  }

  class ErweiterbareListen{
    String bereich;
    String feldbezeichnung;
    List<String> inhalte;
  }

  class CardSetting {
    String page;
    String identifier;
    <<dbColumnDefinition="TEXT">>
    String configuration;
  }

  association [*] CardSetting -> (benutzer) MacocoUser [0..1];

  class Elternzeit { // WZL
    Optional<ZonedDateTime> von;
```

```
  Optional<ZonedDateTime> bis;
  ZahlenWert umfang;
}

class Nebentaetigkeit { // WZL
  Optional<ZonedDateTime> von;
  Optional<ZonedDateTime> bis;
  ZahlenWert umfang;
  Optional<String> arbeitgeber;
  Optional<String> taetigkeitsbereich;
}

class Zulage  { // WZL
  Optional<String> pspelement;
  Optional<String> bezeichnung;
  ZonedDateTime von;
  ZonedDateTime bis;
  Long arbeitgeberbrutto;
  Long arbeitnehmerbrutto;
}

// WZL
enum Geschlecht { NONE(""), WEIBLICH("weiblich"), MAENNLICH("maennlich"), DIVERS("divers"); }

// WZL
// -> DataSource
enum Titel { NONE(""), E1("Apl. Prof. Dr.") , E2("B.Sc.") , E3("Dipl.-Inform.") , E4("Dipl.-Ing."), E5("Dipl.-Ing. (FH)"),
  E6("Dipl.-Ing. (RUS)"), E7("Dipl.-Kff.") , E8("Dipl.-Math."), E9("Dipl.-Phys.") , E10("Dipl.-Wirt. Ing.") , E11("Dr.")  , E12("Dr. phil."),
  E13("Dr. rer. nat."), E14("Dr. rer. pol."), E15("Dr.-Ing."), E16("M.A.") , E17("M.Eng."), E18("M.Sc."), E19("PhD"), E20("Prof. Dr.-Ing."); }

// WZL
// -> DataSource
enum Berufsgruppe { NONE(""), E1("Akademischer Direktor"), E2("Angestellter in der DV"), E3("Auszubildende/r"), E4("Bibl.-Beschaeftiger"),
  E5("Elektriker"), E6("Elektroniker"), E7("Elektrotechniker"), E8("Fachinformatiker"), E9("Fachinformatiker"), E10("Industriemechaniker"),
  E11("Lagerarbeiter"), E12("Mechaniker"), E13("Meister"), E14("Praktikant, SV-frei"), E15("Programmierer"), E16("Stud. Hilfskraft"),
  E17("Techn. Beschaeftigter"), E18("Techn. Zeichner"), E19("Techniker"), E20("Universitaetsprofessor"), E21("Verw. Beschaeftigter"), E22("Werkstoffpruefer"),
```

```
    E23("Wiss. Beschaeftigter"), E24("Wiss. Hilfskraft Bachelor"), E25("Wiss. Hilfskraft Master"); }

  // WZL
  enum Aktion { NONE(""), ANFRAGE_VERSENDET("Anfrage versendet"), ERINNERUNG_VERSENDET("Erinnerung versendet"),
    ZUR_UNTERSCHRIFT_BEIM_VORGESETZTEN("Zur Unterschrift beim Vorgesetzten"),
    BEI_ZHV_VORGELEGT("Bei ZHV vorgelegt"), VON_ZHV_BESTAETIGT("Von ZHV bestaetigt"); }

  // WZL
  // -> DataSource
  enum Vertragsgrundlage { NONE(""), E1("sonst. Befristungen (gleicht Auszubildenden)"), E2("unbefristet"), E3("unbefr. (Beamter aL)"), E4("14 Abs.  2 TzBfG"),E5("Elternzeitvertretung"), E6("Befristung SHK"), E7("Mehrbedarf sonstiger"), E8("Mehrb.  Forschungspr."), E9("2(5)Nr.3WZVG MuS-EZ"
),
    E10("2 WZVG DM (NP)"), E11("2 WZVG KiBetr NP"), E12("WZVG-Postdoc-Quali."), E13("WZVG-N-Gr._Leiter"), E14("WZVG-Wiss.-Quali NP"),
    E15("2 WZVG DM (VP)"),
      E16("2 WZVG KiBetr VP"), E17("WZVG-Promotion"), E18("WZVG-sonst. Wiss-Qual"), E19("Befristung WHB"), E20("Befristung WHK"),
    E21("2 (1) S.2 WZVG nPr"), E22("14(1) Nr.2 TzBfG-ueb"), E23("befr. Arbeitszeitaenderung"), E24("uebernahme nach Ausbildung"); }

  association [1] Person -> (elternzeiten) Elternzeit [*]; // WZL
  association [1] Person -> (nebentaetigkeiten) Nebentaetigkeit[*];  // WZL
  association [1] Person -> (zulagen) Zulage [*];  // WZL

  // WZL Personal Export
  class PersonalExportData {
  }
  association [1] PersonalExportData -> (entries) PersonalExportDataEntry [*];

  class PersonalExportDataEntry {
    String personalnummer;
    Optional<String> vorsatzwort;
    String nachname;
    String vorname;
    ZonedDateTime von;
    ZonedDateTime bis;
    String pspElement;
    Long prozentsatz;
  }

   class PersonalExportEinstellungen {
```

```
    String pspElement;
    List<String> bezeichnungen;
    ZonedDateTime startmonat;
    ZonedDateTime endmonat;
  }

class Organigramm {
  boolean istStabstelle;
  boolean istAktiv;
  String bezeichnung;
}

<<noOrphanRemoval>>
association [0..1] Organigramm (parent) <-> (child) Organigramm [*];

association [1] Organigramm -> (leiter) OrganigrammPersonInfo [*];
association [1] Organigramm -> (stellvertreter) OrganigrammPersonInfo
    [*];
association [1] Organigramm -> (mitarbeiter) OrganigrammPersonInfo [*];

class OrganigrammPersonInfo {
  Optional<ZonedDateTime> von;
  Optional<ZonedDateTime> bis;
  Optional<ZahlenWert> umfang;
}

association [*] OrganigrammPersonInfo -> Person [1];

// Druckeinstellungen
class DruckEinstellungenProjekt {
  Long projektId;
  Integer jahr;
  Integer monat;
  Optional<String> mittelabrufnummer;
  Optional<String> belegnummer;
}

association [1] DruckEinstellungenProjekt -> (
    druckEinstellungenPersonen) DruckEinstellungenPerson [*];

class DruckEinstellungenPerson {
  Long personId;
  Optional<ZahlenWert> stundensatz;
}

class BlacklistedToken {
  <<dbColumnDefinition="TEXT">> // have more than 255 characters
  String token;
  ZonedDateTime addedAt;
  ZonedDateTime expiresAt;
```

```
  }

  class RefreshToken {
    long userId;
    String token;
  }

  // NotificationCenter
  class UserNotification {
    UserNotificationType notificationType;
    String title;

    <<dbColumnDefinition="TEXT">>
    String message;
    ZonedDateTime timeStamp;

    // clickable
    Optional<String> link; // fuer Router

    boolean seen;
    boolean pinned;
  }

  enum UserNotificationType { SUCCESS, INFO, WARNING, DANGER; }

  association [1] MacocoUser -> (notification) UserNotification [*];

  class AccessIdentifier {
  }

  class CommandLog {
    ZonedDateTime timestamp;
    List<String> diff;
    String affectedArea;
    String affectedObject;
  }

  association [1] CommandLog -> (executor) MacocoUser [0..1];
  association [1] CommandLog -> (permission) AccessIdentifier [*];
}
```

Listing A.1: Domain Model of MaCoCo

The Invidas class diagram

```
/* (c) https://github.com/MontiCore/monticore */

package de.monticore.montigem.be.domain;
```

```
import de.monticore.montigem.rte.be.domain.RTEDomain.*;
import de.monticore.montigem.rte.be.dto.IdDTO;
import de.monticore.montigem.be.system.policies.Policies.*;
import java.time.ZonedDateTime;

classdiagram Domain {

  class Datenschutzerklaerung {
    String herstellerName;
    String name;
    ZonedDateTime erstellDBDatum;
    ZonedDateTime aenderungsDBDatum;
    ZonedDateTime gueltigAbDatum;
    int gueltigAbAlter;
    String version;
    String url;
    String volltext;
    String aktualisierungen;
  }

  association [1] Datenschutzerklaerung -> Gebiet [*];
  association [1] Datenschutzerklaerung -> Datenverarbeitung [*];
  association [1] Datenschutzerklaerung -> Datenkategorie [*];
  association [1] Datenschutzerklaerung -> Dateneintrag [*];

  /**************************************************/
   * Gebiete
  /**************************************************/

  class Gebiet {
    String name;
  }

  association [1] Gebiet -> Rechte [*];
  association [1] Gebiet -> Datenschutzbeauftragter [0..1];
  association [1] Gebiet -> Verantwortlicher [*];

  class Datenschutzbeauftragter {
    String name;
    String standort;
  }

  association [1] Datenschutzbeauftragter -> Kontaktdaten [1];

  class Verantwortlicher {
    String name;
    String standort;
  }

  association [1] Verantwortlicher -> Kontaktdaten [1];
```

```
association [1] Verantwortlicher -> Vertreter [0..1];

class Vertreter {
  String name;
}

association [1] Vertreter -> Kontaktdaten [1];

class Kontaktdaten {
  String postadresse;
  List<String> elektrKontakt;
}

class Rechte {
  String gesetz;
  <<dbColumnDefinition="TEXT">>
  String beschreibung;
}

class RechteTypAdapter {
  Rechtetyp rechtetyp;
}

association [1] Rechte -> Rechtewahrnehmung [*];
association [1] Rechte -> RechteTypAdapter [*];

class Rechtewahrnehmung {
  List<String> kontakt;
  <<dbColumnDefinition="TEXT">>
  Optional<String> beschreibung;
}

class Beschwerdewahrnehmung extends Rechtewahrnehmung {
  Optional<String> kontaktAufsichtsbehoerde;
}

/**************************************************/
 * Datenverarbeitung allgemein
/**************************************************/

class Datenverarbeitung {
  Akteur akteur;
  String szenario;
  <<dbColumnDefinition="TEXT">>
  Optional<String> beschreibung;
}

class AkteurOrtAdapter {
  AkteurOrt value;
}
```

```
association [1] Datenverarbeitung -> AkteurOrtAdapter [1..*];
association [1] Datenverarbeitung -> Zweck [1..*];

class Zweck {
  <<dbColumnDefinition="TEXT">>
  Optional<String> beschreibung;
  Zustimmungsart zustimmungsart;
  Optional<String> zustimmungsWiderruf; //TODO
}

association [1] Zweck -> Rechtsgrundlage [1];
association [*] Zweck -> Dateneintrag [*];
association [*] Zweck -> Datenkategorie [*];

class Rechtsgrundlage {
  Rechtsgrundlagetyp rechtsgrundlageTyp;
  <<dbColumnDefinition="TEXT">>
  Optional<String> beschreibung;
}

class Dateneintrag {
  String bezeichnung;
}

class Datenkategorie {
  String name;
}

association [*] Dateneintrag -> Datenkategorie [*];
association datenkategorien [*] Datenkategorie -> (kategorie)
    Datenkategorie [*];

association [*] Dateneintrag -> DatenTagAdapter [*];
association [*] Datenkategorie -> DatenTagAdapter [*];

class DatenTagAdapter {
  DatenTag value;
}

/**************************************************/
 * Spezielle Datenverarbeitungsformen
/**************************************************/


class DatenerhebungUndAufbereitung extends Datenverarbeitung {
  String verarbeitungsform;
  Zeitpunkt zeitpunkt;
}
```

```
class VerarbeitungsformSingleton {
  List<String> values;
}

association [1] DatenerhebungUndAufbereitung ->
    AutomatisierteEntscheidungsfindung [*];

class AutomatisierteEntscheidungsfindung {
  String infoUeberLogik;
  String tragweite;
}

class InfoUeberLogikSingleton {
  List<String> values;
}

class TragweiteSingleton {
  List<String> values;
}

/*************************************************/

class Datenspeicherung extends Datenverarbeitung {
  Optional<String> speicherDauer;
  Optional<String> speicherBisEreignis;
  Optional<String> schutzmassnahme;
  String speicherLand;
}

/*************************************************/

class Datenweitergabe extends Datenverarbeitung {
  String konkreteEmpfaenger;
  boolean auftragsdatenverarbeitung;
  boolean absicht;
  Optional<String> schutzmassnahme;
  Zeitpunkt zeitpunkt;
}

class EmpfaengertypAdapter {
  Empfaengertyp value;
}

association [1] Datenweitergabe -> (empfaengerGebiet) EmpfaengerGebiet
    [*];
association [1] Datenweitergabe -> (empfaengerOrt) AkteurOrtAdapter
    [1..*];
association [1] Datenweitergabe -> (empfaengerTyp) EmpfaengertypAdapter
     [1..*];
```

```
abstract class EmpfaengerGebiet { }

class EWRLandEmpfaenger extends EmpfaengerGebiet {
  List<String> laender;
}

class AngLandEmpfaenger extends EmpfaengerGebiet {
}

association [1] AngLandEmpfaenger -> (angemessenheitsbeschlussLand)
    AngemessenheitsbeschlussLandAdapter [*];

class AngemessenheitsbeschlussLandAdapter {
  AngemessenheitsbeschlussLand value;
}

class GarantieLandEmpfaenger extends EmpfaengerGebiet {
  List<String> laender;
  String garantie;
  String kopieQuelle;
}


/****************************************************
 * Enums
 ****************************************************/

enum DatenTag {
  GesundheitUndFitness,
  Standort,
  Kontaktinformationen,
  Kennungen,              // Passwoerter Nutzername
  Nutzungsdaten,
  Diagnose,
  Finanzinformationen,    // Kreditkartennummer etc.?
  GekaufteArtikel,
  Benutzerinhalte,        // Fotos etc.
  Suchverlauf,
  SonstigeDaten;
}

enum Rechtetyp {
  AUSKUNFT,
  EINSCHRAENKUNG_DER_VERARBEITUNG,
  WIDERSPRUCH_GEGEN_VERARBEITUNG,
  DATENUEBERTRAGBARKEIT,
  WIDERRUF_DER_EINWILLIGUNG,
  BESCHWERDE,
  BERICHTIGUNG,
  LOESCHUNG,
```

```
    SONSTIGES;
  }

  enum Rechtsgrundlagetyp {
    EINWILLIGUNG,
    VERTRAGLICHE_VERPFLICHTUNG,
    GESETZLICHE_VERPFLICHTUNG,
    BERECHTIGTES_INTERESSE,
    ANDERE,
    NICHT_NOTWENDIG;
  }

  enum Empfaengertyp {
    NUTZER,
    HERSTELLER,
    EXTERNE_DATENEMPFAENGER,
    EXTERNE_FREUNDE,
    PLATTFORM_FREUNDE,
    PLATTFORM_NUTZER;
  }

  enum AkteurOrt {
    WEARABLE,
    APP,
    WEBSITE,
    HERSTELLER_INFRASTRUKTUR,
    DRITTANBIETER_INFRASTRUKTUR;
  }

  enum Akteur {
    NUTZER,
    HERSTELLER,
    EXTERNER_DATENBEREITSTELLER,
    EXTERNER_DATENEMPFAENGER;
  }

  enum Zeitpunkt {
    EINMALIG,
    KONTINUIERLICH,
    BEI_AKTIVITAET,
    KEINE_ANGABE;
  }

  enum Zustimmungsart {
    FREIWILLIG,
    FUNKTIONSABHAENGIG_VERPFLICHTEND,
    GRUNDLEGEND_VERPFLICHTEND,
    KEINE_ANGABE;
  }
```

```
  enum AngemessenheitsbeschlussLand {
    ANDORRA,
    ARGENTINIEN,
    KANADA,
    FAEROER_INSELN,
    GUERNSEY,
    ISRAEL,
    ISLE_OF_MAN,
    JAPAN,
    JERSEY,
    NEUSEELAND,
    SCHWEIZ,
    URUGUAY,
    UK;
  }

  class HostSpecificContact {
    String emailAddress;
  }

  class ImpressumInformation {
    <<dbColumnDefinition="TEXT">>
    String impressumHTML;
  }

  class EigeneDatenschutzerklaerung {
    <<dbColumnDefinition="TEXT">>
    String dseHTML;
  }

}
```

Listing A.2: Domain Model of Invidas

A.2 Freemarker Templates

Several Templates used in CD2GUI to create GUI pages for the web application:
GUI-model-template for a Details-Page in GUIDSL v2 syntax

FTL

```
${tc.signature("domainClass", "name", "domainPackage","attributes","roles")}
<#assign AttributeManager = tc.instantiate("cd2gui.util.AttributeManager")>
package cd2gui.models;
//Data classes
import ${domainPackage}.${name};
//GUI models
import arrange.src.lib.gemcard.GemCard;
import arrange.src.lib.gemrow.GemRow;
import basic.src.lib.gemtext.GemText;
import table.src.lib.gemtable.GemTable;
import input.src.lib.gemtextinput.GemTextInput;
import table.src.lib.gemtable.TableTypes.TableColumn;
page ${name}Details(${name} ${name?lower_case}) {
    ${name?uncap_first}DetailsCard@GemCard(
        title = "${name} Details",
        component = ${name?uncap_first}DetailsRow@GemRow(hAlign = "space-between", components = [
            ${name?uncap_first}Id@GemText(value = "Id = " + ${name?lower_case}.gemId)<#if (attributes?size > 0)>,</#if>
            <#list attributes as a>
            <#if AttributeManager.isDerived(a) >
            ${name?uncap_first}_${a.getName()}@GemText(value = "${a.getName()} = " + ${name?lower_case}.${a.getName()})
            <#else >
            ${name?uncap_first}_${a.getName()}@GemTextInput(
                labelText = "${a.getName()}",
                entry = ${name?lower_case}.${a.getName()}
            )
            </#if><#sep>, </#sep>
            </#list>
        ])
    );
    ${name?uncap_first}AssociationCard@GemCard(
        title = "${name} Associations",
        component = ${name?uncap_first}AssociationRow@GemRow(hAlign = "space-between", components = [
            <#list roles as r>
            ${r.getName()?uncap_first}RoleTable@GemTable(
                rows = ${name?lower_case}.${r.getName()},
                columns = [
                    TableColumn("gemId","Id")<#if (AttributeManager.getAssociationAttributes(r, true)?size > 0)>,</#if>
                <#list AttributeManager.getAssociationAttributes(r, true) as ra>
                    TableColumn("_${ra.getName()?lower_case}","${ra.getName()?cap_first}")<#sep>, </#sep>
                </#list>
                ]
            )<#sep>, </#sep>
            </#list>
        ])
    );
}
```

Listing A.3: Template defining the details page

Freemarker Template for GUI-model for overview page

```
<#-- (c) https://github.com/MontiCore/monticore -->
${tc.signature("name", "attributes", "roles", "top", "derivedAttributes", "notDerivedAttributes","isVersionable")}

<#assign astcdClassUtility = tc.instantiate("cd2gui.generator.util.ASTCDClassUtility")>
<#assign associationUtility = tc.instantiate("cd2gui.generator.util.AssociationUtility")>
<#assign attributeUtility = tc.instantiate("cd2gui.generator.util.AttributeUtility")>
<#assign roleUtility = tc.instantiate("cd2gui.generator.util.RoleUtility")>
<#assign stereotypeUtility = tc.instantiate("cd2gui.generator.util.StereotypeUtility")>

<#include "detail-component-parts/imports.ftl">

<#-- If we are not a TOP component/real component, we need the @Component -->
<#include "detail-component-parts/component.ftl">

export class ${name}DetailsComponent${top} extends ${name?lower_case?cap_first}DetailsComponent${top}TOP {

    protected router: Router;

<#include "detail-component-parts/constructor.ftl">

    <#list attributes as a>
        <#assign attr=a.getName()>
        <#if !stereotypeUtility.isNonEditable(a) && !attributeUtility.isList(a)>
            ${attr}ListObject
        </#if>
    </#list>

    public ngOnInit() {
        super.ngOnInit();
    }

<#include "detail-component-parts/association_table_input_filter.ftl">

<#include "detail-component-parts/derived_attributes.ftl">

<#list notDerivedAttributes as a>
    <#if !stereotypeUtility.isNonEditable(a) && !attributeUtility.isList(a)>
    <#assign attr=a.getName()>
    public cancelSave_${attr}():void {
```

```
        this.edit_${attr}Mode = false;
    }

    <#include "detail-component-parts/edit_save_method.ftl">

    </#if>
</#list>


    private updateAssociation(): void {
        let fullDto = new ${name}FullDTO(this.dto);
        fullDto.id = this.id;
    <#list roles as role>
      fullDto.${role.getName()} = this.dto.${role.getName()};
    </#list>
        fullDto.update(this.commandManager).then(() => {
            this.commandManager = new CommandManager(this.
                _commandRestService);
            this.initAllCommands();
        });
        this.sendCommands();
    }

//--------------------------------------------------------
//---ROLE METHODS-----------------------------------------
//--------------------------------------------------------

<#list roles as role>
    <#include "detail-component-parts/role_methods.ftl">
</#list>

<#if isVersionable == true>
  public release_${name}(): void {
  this.router.navigateByUrl("dashboard/release/" + this.id)
  }
</#if>
}
```

Listing A.4: Freemarker Template for Overview Page

A.3 GUI-models

GUI-model for a Overview-Page in GUIDSL v1 syntax

GUI-DSL v1

```
/* (c) MaCoCo, ein RWTH Aachen projekt */
webpage FinanzenOverview( all KontenOverviewProvider provider,
```

```
                          all KontenOverview ko,
                          all BudgetInfo bi,
                          enum Vergabeverordnung vo) {

row (r) {
  balances <bi {
    box "Gesamtbudget", <gesamtBudget, default;
    box "Ausgaben", <ausgaben, default;
    box "Planausgaben", <planBudget, default;
    box "Restbudget", <restBudget, success;
  }
}

column{
    card "card1" {
      head{
        row (stretch) {
          textoutput {"Ubersicht Konten"}
          row (r) {
            button "Zahlungen erfassen"{
              styleclass:"blue-green-transition"
              click -> navigateToZahlungen()
            }
            button "Konto hinzufugen"{
              styleclass:"blue-green-transition"
              click -> addKonto()
            }
            helpbutton Wiki "https://macoco.rwth-aachen.de/w/index.php/
                Konten%C3%BCbersichtsseite"
          }
        }
      }
      body{
        row(50%, spacebelow) {
          container(220px) {
            multi dropdown  {
              placeholder: "Kontoart"
              input: <kontoartFilter
            }
          }
          container(350px) {
            multi dropdown  {
              placeholder: "Organigrammstufe"
              input: <organigrammStufeFilter
            }
          }
          container(350px) {
            multi dropdown  {
              placeholder: "Fachliche Verantwortung"
              input: <fachlicheVerantwortungFilter
```

```
            }
          }
        }
        datatable "finanzenOverviewDatatable" {
          references {
            rowClass: rowClass
            selected : selected
            rightClickMenu : contextMenu
            activeRow: activeRow
            rowHeight: 29
            exportname: Uebersicht_Konten
          }
          methods {
            select -> onSelect($event)
          }
          conditions {
            skipNotification
            groupable
          }
          allowBatchMode : Konto istAktiv
          rows <ko.kontenOverviewEntries {
            column "PSP-Element"
                , < pspElement                         , 120;
            column "Name"
                , < name                               , 120;
            column "Typ"
                , < typ                                , 100;
            column "Projektart"
                , <projektArt                          , 40;
            column "Laufzeitbeginn"
                , date(< laufzeitBeginn)               , 80, hidden;
            column "Laufzeitende"
                , date(< laufzeitEnde)                 , 80;
            column "Laufzeit"
                , < laufzeitInMonate                   , 40;
            column "Gesamtbudget"
                , euro(<budget[0].gesamtBudget)        , 80;
            column "Ausgaben"
                , euro(<budget[0].ausgaben)            , 80;
            column "Ausgaben Wissenschaftliche Mitarbeiter/innen"
                , euro(<budget[0].ausgabenWiMis)       , 80, hidden;
            column "Ausgaben Hilfskrafte"
                , euro(<budget[0].ausgabenHiWis)       , 80, hidden;
            column "Verplant"
                , euro(<budget[0].planBudget)          , 80;
            column "Saldo"
                , euro(<budget[0].restBudget)          , 80;
            column "Einnahmen"
                , euro(<einnahmen)                     , 80, hidden;
            column "Kontostand"
```

```
                , euro(<budget[0].kontostand)         , 80, hidden;
            column "Offene Abgleiche"
                , <nichtAbgeglicheneBuchungen         , 80;
            column "Fachliche Verantwortung"
                , <fachlicheVerantwortung             , 80, hidden;
            column "Organigrammstufe"
                , <organigrammStufe                   , 80, hidden;
            column "Vergabeverodnung"
                , <vergabeVerordnung                  , 100, hidden;
            column "Label"
                , < labels                            , 100;
            column "anderungsdatum"
                , date(< aenderungsdatum)             , 100,hidden,
                disabled;
          }
        }
      }
    }
  }
  contextmenu "contextMenu" {
    group {
      entry if (isSingleSelection() && isAktiv(item)) "Bearbeiten: Neuer
          Tab" -> navigateToFormular($event, true)
      entry if (isSingleSelection() && isAktiv(item)) "Bearbeiten: Hier
          offnen" -> navigateToFormular($event, false)
      entry if (!isInBatchMode() && isAktiv(item)) "Deaktivieren" ->
          deactivateAccount($event)
      entry if (!isInBatchMode()) "Aktivieren" -> activateAccount($event)
      entry if (!isInBatchMode()) "Loschen" -> deleteKonto($event)
      entry if (!isInBatchMode()) "Projekt erstellen" -> createProjekt(
          $event)
    }
    group {
      entry if (noSelection()) "Sie haben keine Eintrage selektiert!" ->
          emptyFunction($event)
      entry if (!noSelection()) "Neues Konto anlegen" ->
          addKontoViaContextMenu()
    }
    group if (isSingleSelection()) {
      entry "Detailansicht: Neuer Tab" -> navigateToDetails($event, true)
      entry "Detailansicht: Hier offnen" -> navigateToDetails($event,
          false)
      entry "Buchung anlegen: Neuer Tab" -> navigateToBuchungen($event,
          true)
      entry "Buchung anlegen: Hier offnen" -> navigateToBuchungen($event,
           false)
      entry "Overheads: Neuer Tab" -> navigateToOverheads($event, true)
      entry "Overheads: Hier offnen" -> navigateToOverheads($event, false
          )
      entry getGeschaeftsvorgng(item) + " erfassen" -> navigateToPage(
```

```
        $event)
      entry if (isZuweisung(item))"Stellenzuweisung erfassen: Neuer Tab"
        -> navigateToStellenzuweisungen($event, true)
      entry if (isZuweisung(item))"Stellenzuweisung erfassen: Hier offnen
        " -> navigateToStellenzuweisungen($event, false)
    }
    group {
      entry if (isBatchSelected()) "Loschen" -> batchDelete($event)
      entry if (isBatchSelected()) "Aktivieren" -> batchActivate($event)
      entry if (isBatchSelected()) "Deaktivieren" -> batchDeactivate(
        $event)
      entry if (isBatchSelected()) "Plankonto" -> batchSetPlankonto(
        $event)
      entry if (isBatchSelected()) "Kein Plankonto" ->
        batchUnsetPlankonto($event)
      entry if (isBatchSelected()) "Projekte erstellen" ->
        batchCreateProjekt($event)
    }
  }

}
```

Listing A.5: GUI-DSL v1 Model for an Overview page in MaCoCo

GUI-model-Output by CD2GUI for a Overview-Page in GUIDSL v2 syntax

```
// ...

page personOverview(List<person> person) {
    // ...
    personOverviewCard@GemCard(
        title = "Person Overview",
        components = [
            // ...
            personOverviewRow@GemRow(hAlign = "space-between", components
                 = [
            personInstancesTable@GemTable(
                rows = person,
                columns = [
                    TableColumn(person.id, "Id"),
                    TableColumn(person.name, "Name")
                    // ...
                ]
            ]
            )
        ])
    );
}
```

Listing A.6: Overview Page in GUIDSL v2 syntax

A.4 Few-Shot Learning Example Files

We use several models as examples to fine-tune the large language models we use. In the following we list the models we used as examples:

A.4.1 MontiArc

```
/* (c) https://github.com/MontiCore/monticore */
package elevator;

component Elevator {

  port <<sync>> in Boolean req1, req2, req3, req4,
       <<sync>> out Integer clear;
  port <<sync>> in Boolean at1, at2, at3, at4,
       <<sync>> out Boolean open, close, up, down,
       <<sync>> in Boolean isOpen, isClosed, isObstacle;

  Controller ctrl;

  req1 -> ctrl.req1;
  req2 -> ctrl.req2;
  req3 -> ctrl.req3;
  req4 -> ctrl.req4;

  ctrl.clear -> clear;

  at1 -> ctrl.at1;
  at2 -> ctrl.at2;
  at3 -> ctrl.at3;
  at4 -> ctrl.at4;

  isOpen -> door.isOpen;
  isClosed -> door.isClosed;
  isObstacle -> door.isObstacle;

  Door door;

  door.open -> open;
  door.close -> close;
  door.closed -> ctrl.isClosed;

  Lift lift;

  lift.up -> up;
  lift.down -> down;
```
MA

```
  ctrl.door -> door.cmd;
  ctrl.lift -> lift.cmd;

}
```

Listing A.7: Elevator MontiArc Model

```
/* (c) https://github.com/MontiCore/monticore */
package bumperbot;

component BumperBot {

  Ultrasonic sensor;
  Motor leftMotor;
  Motor rightMotor;
  BumpControl controller;
  Timer timer (1000);
  Logger logger;

  sensor.distance   -> controller.distance;
  controller.right  -> rightMotor.cmd;
  controller.left   -> leftMotor.cmd;
  controller.timer  -> timer.cmd;
  timer.signal      -> controller.signal;
  controller.speed  -> leftMotor.speed;
  controller.speed  -> rightMotor.speed;
  controller.log    -> logger.message;

}
```

MA

Listing A.8: BumperBot MontiArc Model

A.4.2 Sequence Diagrams

```
/* (c) https://github.com/MontiCore/monticore */

package examples.correct.lecture;

complete sequencediagram example_pretty {

  complete kupfer912: Auction;
  bidPol: BiddingPolicy;
  timePol: TimingPolicy;
```

SD

```
  a -> b : methodenname(arg);
  f -> BidMessage bm = new BidMessage(...);

  class A -> b : trigger test();

  a -> b : static test() {
    a -> b : test();
    b <- a : test();
  }
}
```

Listing A.9: Example Input for MontiCore sequence diagram syntax.

```
/* (c) https://github.com/MontiCore/monticore */

package correct;

sequencediagram allGrammarElements {

  // Interacting objects
  (c) o: Order;
  c: Customer;

  // Offer -> Production
  o -> c : trigger sendConfirmation();
  o <- c : return;
  o -> o : orderParts();
  assert state == Production;

  // Production -> Shipping
  o -> o : trigger shipItems();
  assert state == Shipping;

  // Shipping -> Payment
  o -> c : sendInvoice(sum);
  o <- c : return;
  assert state == payment;

  // Payment -> Complete
  assert state == complete;
  o -> Mail m = new Mail();
}
```

Listing A.10: Example Input for MontiCore sequence diagram syntax.

```
/* (c) https://github.com/MontiCore/monticore */

/*
 * Example in documentation.
 */
package examples.correct;

sequencediagram size {

  kupfer912:Auction;
  theo:Person;

  kupfer912 -> BidMessage bm = new BidMessage(...);
  let int m = theo.messages.size;
  kupfer912 -> theo : sendMessage(bm);
  theo -> kupfer912 : return;
  assert m + 1 == theo.messages.size;
}
```

Listing A.11: Example Input for MontiCore sequence diagram syntax.

```
/* (c) https://github.com/MontiCore/monticore */                    SD

/*
 * Example in documentation.
 */
package examples.correct;

sequencediagram bid {

  kupfer912:Auction;
  bidPol:BiddingPolicy;
  timePol:TimingPolicy;
  theo:Person;

  kupfer912 -> bidPol : validateBid(bid) {
    bidPol -> kupfer912 : return BiddingPolicy.OK;
  }
  kupfer912 -> timePol : newCurrentClosingTime(kupfer912,bid) {
    timePol -> kupfer912 : return t;
  }
  assert t.timeSec == bid.time.timeSec + extensionTime;
  let int m = theo.messages.size;
  kupfer912 -> theo : sendMessage(bm) {
    theo -> kupfer912 : return;
  }
```

```
  assert m + 1 == theo.messages.size;
}
```

Listing A.12: Example Input for MontiCore sequence diagram syntax.

A.4.3 Feature Diagrams

```
/* (c) https://github.com/MontiCore/monticore */
package fddiff;

featurediagram car1 {
  car -> engine & locking?;
  engine -> electric | gas;
  locking -> keyless ^ phone ^ fingerprint;
}
```

FD

Listing A.13: Example Input for MontiCore feature diagram syntax.

```
/* (c) https://github.com/MontiCore/monticore */
package fdvalid;

featurediagram PhoneComplex {
  Phone -> Memory & OS & Camera? & Screen;
  Memory -> Internal & External?;
  Internal -> [1..2] of {Small, Medium, Large};
  OS -> iOS ^ Android;
  Screen -> Flexible | FullHD;
  External ? Flexible requires Android : iOS && Android;
}
```

FD

Listing A.14: Example Input for MontiCore feature diagram syntax.

```
/* (c) https://github.com/MontiCore/monticore */
package fdvalid;

featurediagram CarNavigation {
  CarNavigation -> Display & GPS & PreinstalledMaps? & Memory;
  CarNavigation -> VoiceControl ^ TouchControl;
  Memory -> Small ^ Medium ^ Large;
  Display -> SmallScreen | LargeScreen;
```

FD

```
  PreinstalledMaps -> [1..3] of {Europe, NorthAmerica, SouthAmerica, Asia
      , Africa};
  TouchControl requires LargeScreen;
  SmallScreen excludes TouchControl;
  (Europe && NorthAmerica && Asia) requires (Large || Medium);
}
```

Listing A.15: Example Input for MontiCore feature diagram syntax.

```
/* (c) https://github.com/MontiCore/monticore */

package fdvalid;

featurediagram Phone {

  Phone -> Memory & OS & Camera? & Screen;
  Memory -> Internal & External?;
  Internal -> [1..2] of {Small, Medium, Large};
  OS -> iOS ^ Android;
  Screen -> Flexible | FullHD;

  iOS excludes External;
  Flexible requires Android;

}
```

Listing A.16: Example Input for MontiCore feature diagram syntax.

A.4.4 Examtask

Exam Task for the modelling of an E-Bike:
Generate a class diagram of an EBike according to these specifications:
The EBike is composed of a frame (made out of steel), a drive system, and a controller. Two wheels are inserted into each frame. The drive system is composed of a motor. Each EBike can be connected to a removable battery. The battery has a stored energy measured in Watt-hours (Wh). The controller can be in one of three states: On, Off, and Charging. It also controls the battery, if one is connected, and commands the drive system. The company plans two different variants of the controller, a basic controller, and an advanced controller. The advanced controller should be able to estimate the next Date the bike should be inspected for maintenance.

A.5 CD4A-Models containing Syntax Errors

The following models contain simple syntax errors. The LLM was task to fix these models based on parser feedback.

CD4A

```
classdiagram PingPongGame {

  class PingPongTable {}

  class Player
    String name;

  class Ball {}

  class PingPongGame {
    Player player1;
    Player player2;
    Ball ball;
    Referee referee;
  }

  association [1] PingPongGame -> (table) PingPongTable [1];
  association [1] PingPongGame -> (ball) Ball [1];
}
```

Listing A.17: CD4A model defining a class, but leaving out brackets

CD4A

```
class PingPongGame {

  class PingPongTable {}

  class Player {
    String name;
  }

  class Ball {}

  class PingPongGame {
    Player player1;
    Player player2;
    Ball ball;
    Referee referee;
  }

  association [1] PingPongGame -> (table) PingPongTable [1];
  association [1] PingPongGame -> (ball) Ball [1];
```

```
}
```

Listing A.18: CD4A class diagram not starting with the keyword 'classdiagram'.

CD4A

```
classdiagram PingPongGame {

  class PingPongTable {}

  class Player{
    String name;
  }

  class Ball {}

  class PingPongGame {
    Player player1;
    Player player2;
    Ball ball;
    Referee referee;
  }

  association [1] PingPongGame --|> (table) PingPongTable [1];
  association [1] PingPongGame --|> (ball) Ball [1];
}
```

Listing A.19: CD4A class diagram using bad notation to denote an association.

CD4A

```
classdiagram PingPongGame {

  class PingPongTable {}

  class Player{
    String name;
  }

  class Ball {}

  class PingPongGame {
    Player player;
    Player player;
    Ball ball;
    Referee referee;
  }

  association [1] PingPongGame -> (table) PingPongTable [1];
```

```
  association [1] PingPongGame -> (ball) Ball [1];
}
```

Listing A.20: CD4A class diagram defining two attributes with the same name ('player').

Appendix B

Diagram and Listing Tags

Tag	Description
CD	Class Diagram
CD4A	Class Diagram for Analysis Diagram
Java	Java Source Code
MCG	MontiCore Grammar
MCL	MontiCore Languages

Table B.1: Explanation of the used tags in listings and figures.

Stereotype	Description
«EXT»	External elements
«GEN»	Generated elements
«HC»	Handcoded elements
«RTE»	Run-time Environment elements
«RT-IF»	Run-time Infrastructure elements

Table B.2: Explanation of the used stereotypes in listings and tags.

Appendix C

Screenshots Screenshots of the Management Cockpit for Controlling (MaCoCo). The screenshots are taken August 2022 of test data (TestDB) on the production system.

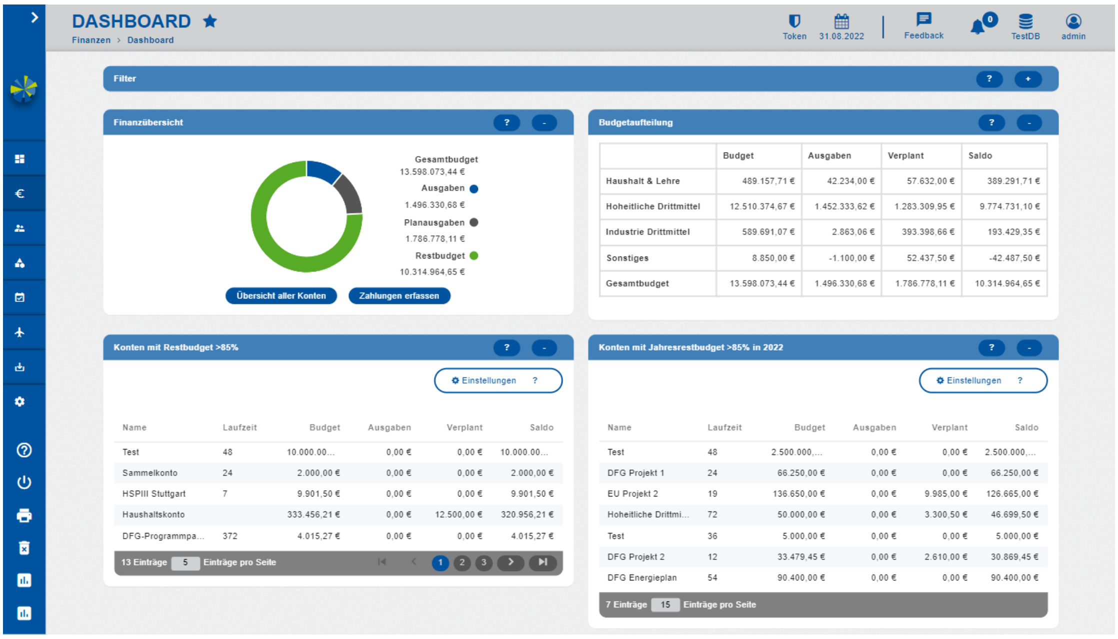


Figure C.1: Finance Dashboard showing charts, filtered tables and key performance indicators (*cf.* CD2GUI Section 6.2.1)

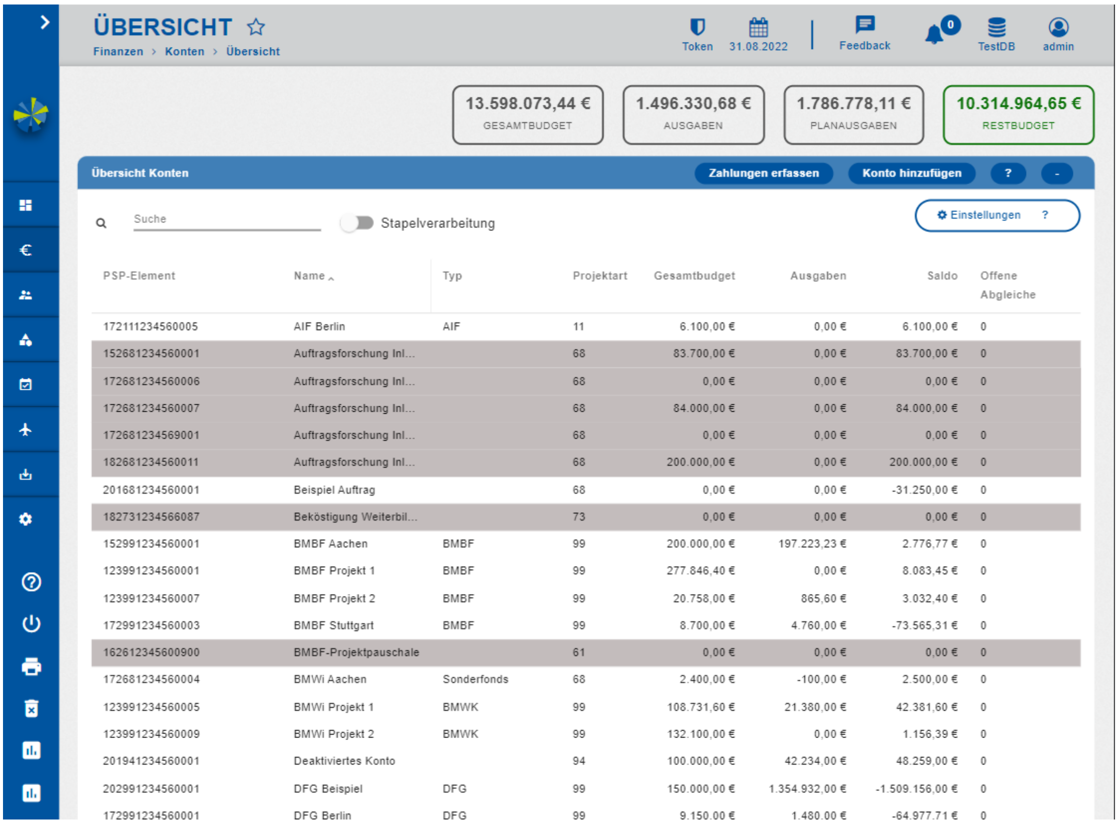


Figure C.2: Overview page showing all bank accounts within the system (*cf.* CD2GUI Section 6.2.2)

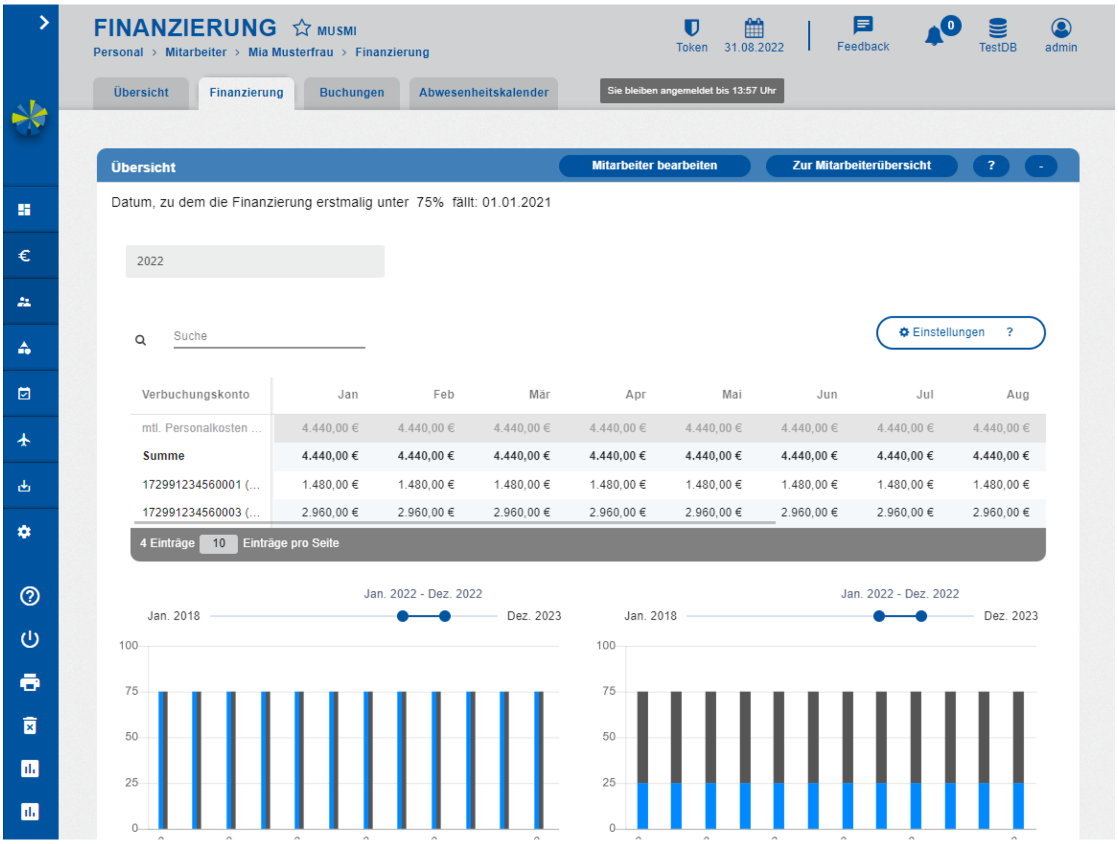


Figure C.3: Detailed Staff page (*cf.* CD2GUI Section 6.2.3)

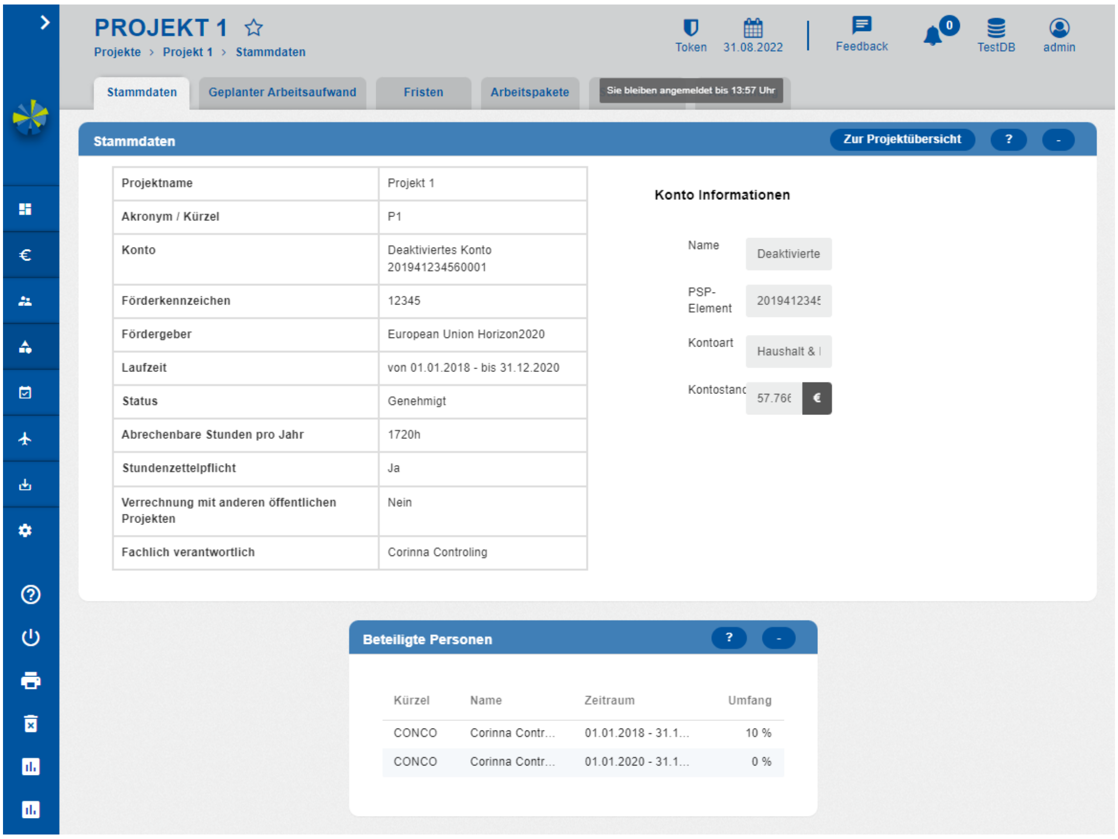


Figure C.4: Detailed Project page (*cf.* CD2GUI Section 6.2.3)

List of Figures

Listings

List of Tables

Glossary

AST An Abstract Syntax Tree is a tree-like data structure representing the hierarchical structure of source code or a model, capturing its syntax and semantics while abstracting away from the specific textual representation. 30, 32, 162, 184, 185, 187–189

BPMN Business Process Model and Notation 60

CD Class Diagram xiv, 43, 53, 56, 60, 62, 67, 125, 161, 219, 225, 348, 352

CD2GUI Class Diagram to GUI Model transformer. A tool developed to enable rapid prototyping of data-centric information systems (c.f. Chapter 6). vii, ix, xiii, 10, 15, 19–25, 115, 116, 118, 121, 123, 127, 128, 130, 133–141, 143, 146–152, 159, 161, 208, 215, 235–237, 246, 247, 249, 258, 261, 262, 265–267, 270, 291, 324, 331, 343–347, 349, 350, 352, 353, 357, 362

CD4A Class Diagram for Analysis xiv, 15, 21, 33–38, 43, 44, 46, 50, 53–58, 60, 64, 67, 80, 100, 111, 112, 135, 137, 140, 148, 183, 184, 206, 230, 239, 244, 256–258, 264, 265, 291, 347–350, 355, 361

ChatGPT ChatGPT is an advanced language model developed by OpenAI, based on the GPT architecture, designed to generate human-like text and engage in conversations by understanding context, providing relevant information, and offering creative responses, enabling a wide range of applications in areas such as customer support, content creation, and virtual assistance. 55, 61, 64, 65, 67, 68, 70

CRUD Create, Read, Update, Delete: The four primary access operations for data. 139

DAO A Data Access Object is a design pattern that abstracts and encapsulates data access and storage operations, providing a consistent interface for interacting with different data sources, such as databases or APIs 124–126, 167, 171, 172, 175, 179, 193–195, 351

domain expert The Domain Expert, also known as Subject Matter Expert, is a person that has expert knowledge of a specific domain e.g.: A watchmaker is a domain expert on watches. 4, 5, 8, 10, 13, 19–21, 24, 25, 44, 52, 53, 64, 208, 210, 250, 255, 348

DSL A Domain Specific Language is a specialized programming language tailored to a particular application domain, designed to simplify tasks, improve expressiveness, and enhance productivity within that specific context. 4, 5, 7, 12, 23, 30, 33, 37, 43, 44, 46, 54, 55, 58, 60, 62, 68, 70, 71, 74, 80–82, 100, 111, 184, 211, 230, 234, 237, 241, 250, 251, 256–258, 262, 265, 268, 269, 349

DTO A Data Transfer Object is a lightweight, serializable object used to encapsulate and transfer data between different layers or components of a software system 167, 175, 176, 178–180, 195, 206, 351, 358

EIS An Enterprise Information System is a software solution that manages, integrates, and processes an organization's data, facilitating decision-making, communication, and collaboration across various business functions and departments. 124, 125, 194, 349

GUI Graphical User Interface 236, 238, 242, 353

GUIDSL DSL for Graphical User Interfaces xiv, 15, 17, 22, 43, 76, 79–83, 85, 87, 93–95, 97, 99–105, 107, 109, 119, 121, 122, 130–132, 143, 152–155, 187, 206, 211, 215, 220, 232–234, 237, 242, 244, 257, 258, 261, 264, 266, 268, 291, 324, 327, 331, 348, 349, 352, 356–359, 362

HTML Hypertex Markup Language 80, 190, 204, 358

Information System A Data-Centric Information System is a type of information system primarily focused on the collection, storage, processing, and management of data as its central asset. In this system, data is treated as a core resource around which applications, processes, and services are built. The emphasis is on ensuring data integrity, availability, and accessibility, often utilizing a centralized database or data repository to support decision-making and operational tasks. 5

LCDP Low Code Development Platform: A software framework that allows rapid application development with minimal hand-coding, enabling non-technical users to create applications through visual interfaces or tooling instead of traditional programming methods. xvi, 2, 235, 241, 242, 251–254

LLM A Large Language Model is a type of artificial intelligence model that uses deep learning techniques to understand natural language, generate human-like text, and perform a range of language-related tasks, such as language translation, question-answering, and text summarization. xiv, 2, 5, 7, 21, 24, 43–46, 48, 49, 51–55, 57, 58, 61, 62, 64, 65, 67–71, 73–77, 145, 250, 251, 254, 255, 262, 265, 270, 347, 362

LLM4CD Large Language Model Based Transformation to Class Diagram. A transformer that converts natural language based informal specifications into CD4A class diagrams (c.f. **??**). ix, xiii, 10, 15, 19–21, 25, 26, 70, 127, 161, 208, 237, 249, 250, 261, 262, 265, 270, 291, 347

MaCoCo Management Cockpit for Controlling, is a software solution that provides an integrated, model-driven approach to manage, monitor, and control complex organizational processes and systems, enabling efficient decision-making and evaluation. ix, xvi, xvii, 1, 2, 7, 10, 16, 17, 24, 50, 81, 82, 87, 88, 93, 95, 97, 99, 121, 125, 138, 163, 180, 195, 205–220, 223–227, 229–234, 238, 249, 256, 259, 260, 262, 264, 269, 291, 331, 343, 347–349, 352, 353, 359, 362

MDE Model-Driven Engineering is a software development approach that emphasizes the use of models, abstractions, and automated tools to design, analyze, and generate software systems 251

MDSE Model-Driven Software Engineering: An approach to software development that emphasizes the use of models and automated tools to facilitate the design, analysis, and generation of software systems. xiv, 38, 43, 71, 73, 75, 77, 234

MontiCore MontiCore is a language workbench that enables the development of domain-specific languages (DSLs) and supports model-driven engineering (MDE) through the creation, composition, and processing of textual models, facilitating code generation and analysis in a modular and extensible framework. xiii, xiv, xvii, 10, 15, 27, 29–33, 38, 43, 51, 54, 60, 64, 66, 71, 73, 74, 81, 82, 102, 149, 156, 162, 184, 203, 249, 251, 255–257, 291, 334–337, 347, 348, 350, 359, 361

MontiGem MontiCore based Generator for Enterprise Management is a comprehensive, model-driven software engineering framework designed to facilitate the development, integration, and maintenance of enterprise management applications through the use of domain-specific languages, model transformations, and automated code generation. vii, ix, xiii, xv, xvi, 10, 16, 17, 19, 20, 23, 25, 28, 38, 81, 99, 112, 127, 130, 140, 161–168, 170–180, 182–188, 190–206, 211–214, 217, 230, 231, 233–245, 249, 254, 256, 258, 261, 262, 264, 267–269, 291, 350–353, 357

NCDP No-Code Development Platform 253

OCL Object Contraint Language 23, 181, 231, 256

RBAC Role Based Access Control: See Section 6.5.1 124, 138–141, 143, 144, 151, 357

root class diagram Within this Work the class Diagram used to derive further gui-models and class diagrams. It is the central artifact used to generate a web application. 21, 23, 120, 123, 124, 128, 140, 150, 151, 167, 210, 234, 265

RTE Run-Time Environment 167, 202

SQL Structured Query Language 80

TRL Technology Readiness Levels (TRLs) are a scale used to assess the maturity of evolving technologies, ranging from initial concept (TRL 1) to fully operational and tested systems (TRL 9). 8, 9, 198, 205, 270

UMLP Unified Modeling Language (UML) is a standardized, general-purpose visual modeling language used in software engineering to represent, analyze, and document the structure, behavior, and interactions of software systems, providing a common notation and set of diagrams that facilitate communication, collaboration, and understanding among developers, analysts, and stakeholders. 70, 183, 250

Related Interesting Work from the SE Group, RWTH Aachen

The following section gives an overview of related work done at the SE Group, RWTH Aachen. More details can be found on the website `www.se-rwth.de/topics/` or in [HMR+19]. The work presented here mainly has been guided by our mission statement:
Our mission is to define, improve, and industrially apply *techniques*, *concepts*, and *methods* for *innovative and efficient development* of software and software-intensive systems, such that *high-quality* products can be developed in a *shorter period of time* and with *flexible integration* of *changing requirements*. Furthermore, we *demonstrate the applicability* of our results in various domains and potentially refine these results in a domain specific form.

Agile Model Based Software Engineering

Agility and modeling in the same project? This question was raised in [Rum04c]: “Using an executable, yet abstract and multi-view modeling language for modeling, designing and programming still allows to use an agile development process.”, [JWCR18] addresses the question of how digital and organizational techniques help to cope with the physical distance of developers and [RRSW17] addresses how to teach agile modeling.
Modeling will increasingly be used in development projects if the benefits become evident early, e.g with executable UML [Rum02] and tests [Rum03]. In [GKR+06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum11, Rum12] and [Rum16, Rum17], the UML/P, a variant of the UML especially designed for programming, refactoring, and evolution is defined.
The language workbench MontiCore [GKR+06, GKR+08, HKR21] is used to realize the UML/P [Sch12]. Links to further research, e.g., include a general discussion of how to manage and evolve models [LRSS10], a precise definition for model composition as well as model languages [HKR+09], and refactoring in various modeling and programming languages [PR03]. To better understand the effect of an agile evolving design, we discuss the need for semantic differencing in [MRR10].
In [FHR08] we describe a set of general requirements for model quality. Finally, [KRV06] discusses the additional roles and activities necessary in a DSL-based software development project. In [CEG+14] we discuss how to improve the reliability of adaptivity through models at runtime, which will allow developers to delay design decisions to runtime adaptation. In [KMA+16] we have also introduced a classification of ways to reuse modeled software components.

Artifacts in Complex Development Projects

Developing modern software solutions has become an increasingly complex and time consuming process. Managing the complexity, the size, and the number of artifacts developed and used during a project together with their complex relationships is not trivial [BGRW17].
To keep track of relevant structures, artifacts, and their relations in order to be able, e.g., to evolve or adapt models and their implementing code, the *artifact model* [GHR17, Gre19] was introduced. [BGRW18] and [HJK+21] explain its applicability in systems engineering based on MDSE projects and [BHR+18] applies a variant of the artifact model to evolutionary development, especially for CPS.
An artifact model is a meta-data structure that explains which kinds of artifacts, namely code

files, models, requirements files, etc. exist and how these artifacts are related to each other. The artifact model, therefore, covers the wide range of human activities during the development down to fully automated, repeatable build scripts. The artifact model can be used to optimize parallelization during the development and building, but also to identify deviations of the real architecture and dependencies from the desired, idealistic architecture, for cost estimations, for requirements and bug tracing, etc. Results can be measured using metrics or visualized as graphs.

Artificial Intelligence in Software Engineering

MontiAnna is a family of explicit domain specific languages for the concise description of the architecture of (1) a neural network, (2) its training, and (3) the training data [KNP+19]. We have developed a compositional technique to integrate neural networks into larger software architectures [KRRW17] as standardized machine learning components [KPRS19]. This enables the compiler to support the systems engineer by automating the lifecycle of such components including multiple learning approaches such as supervised learning, reinforcement learning, or generative adversarial networks.
For analysis of MLOps in an agile development, a software 2.0 artifact model distinguishing different kinds of artifacts is given in [AKK+21].
According to [MRR11g] the semantic difference between two models are the elements contained in the semantics of the one model that are not elements in the semantics of the other model. A smart semantic differencing operator is an automatic procedure for computing diff witnesses for two given models. Such operators have been defined for Activity Diagrams [MRR11d], Class Diagrams [MRR11b], Feature Models [DKMR19], Statecharts [DEKR19], and Message-Driven Component and Connector Architectures [BKRW17, BKRW19]. We also developed a modeling language-independent method for determining syntactic changes that are responsible for the existence of semantic differences [KR18a].
We apply logic, knowledge representation, and intelligent reasoning to software engineering to perform correctness proofs, execute symbolic tests, or find counterexamples using a theorem prover. We have defined a core theory in [BKR+20], which is based on the core concepts of Broy's Focus theory [RR11, BR07], and applied it to challenges in intelligent flight control systems and assistance systems for air or road traffic management [KRRS19, KMP+21, HRR12].
Intelligent testing strategies have been applied to automotive software engineering [EJK+19, DGH+19, KMS+18], or more generally in systems engineering [DGH+18]. These methods are realized for a variant of SysML Activity Diagrams (ADs) and Statecharts.
Machine Learning has been applied to the massive amount of observable data in energy management for buildings [FLP+11, KLPR12] and city quarters [GLPR15] to optimize operational efficiency and prevent unneeded CO_2 emissions or reduce costs. This creates a structural and behavioral system theoretical view on cyber-physical systems understandable as essential parts of digital twins [RW18, BDH+20].

Generative Software Engineering

The UML/P language family [Rum12, Rum11, Rum16] is a simplified and semantically sound derivate of the UML designed for product and test code generation. [Sch12] describes a flexible generator for the UML/P, [Hab16] for MontiArc is used in domains such as cars or robotics [HRR12], and [AMN+20a] for enterprise information systems based on the MontiCore language

workbench [KRV10, GKR+06, GKR+08, HKR21].
In [KRV06], we discuss additional roles necessary in a model-based software development project. [GKR+06, GHK+15, GHK+15a] discuss mechanisms to keep generated and handwritten code separated. In [Wei12, HRW15, Hoe18], we demonstrate how to systematically derive a transformation language in concrete syntax and, e.g., in [HHR+15, AHRW17] we have applied this technique successfully for several UML sub-languages and DSLs.
[HNRW16] presents how to generate extensible and statically type-safe visitors. In [NRR16], we propose the use of symbols for ensuring the validity of generated source code. [GMR+16] discusses product lines of template-based code generators. We also developed an approach for engineering reusable language components [HLN+15, HLN+15a].
To understand the implications of executability for UML, we discuss the needs and the advantages of executable modeling with UML in agile projects in [Rum04c], how to apply UML for testing in [Rum03], and the advantages and perils of using modeling languages for programming in [Rum02].

Unified Modeling Language (UML) & the UML-P Tool

Starting with the early identification of challenges for the standardization of the UML in [KER99] many of our contributions build on the UML/P variant, which is described in the books [Rum16, Rum17] and is implemented in [Sch12].
Semantic variation points of the UML are discussed in [GR11]. We discuss formal semantics for UML [BHP+98] and describe UML semantics using the "System Model" [BCGR09], [BCGR09a], [BCR07b] and [BCR07a]. Semantic variation points have, e.g., been applied to define class diagram semantics [CGR08]. A precisely defined semantics for variations is applied when checking variants of class diagrams [MRR11e] and object diagrams [MRR11c] or the consistency of both kinds of diagrams [MRR11f]. We also apply these concepts to activity diagrams [MRR11a] which allows us to check for semantic differences in activity diagrams [MRR11d]. The basic semantics for ADs and their semantic variation points are given in [GRR10].
We also discuss how to ensure and identify model quality [FHR08], how models, views, and the system under development correlate to each other [BGH+98b], and how to use modeling in agile development projects [Rum04c], [Rum03] and [Rum02].
The question of how to adapt and extend the UML is discussed in [PFR02] describing product line annotations for UML and more general discussions and insights on how to use meta-modeling for defining and adapting the UML are included in [EFLR99a], [FEL+98] and [SRVK10].
The UML-P tool was conceptually defined in [Rum16, Rum17, Rum12, Rum11], got the first realization in [Sch12], and is extended in various ways, such as logically or physically distributed computation [BKRW17a]. Based on a detailed examination [JPR+22], insights are also transferred to the SysML 2.

Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use than general-purpose programming languages but need appropriate tooling. The MontiCore language workbench [GKR+06, KRV10, Kra10, GKR+08, HKR21] allows the specification of an integrated abstract and concrete syntax format [KRV07b, HKR21] for easy development. New languages and tools can be defined in modular forms [KRV08, GKR+07, Voe11, HLN+15, HLN+15a,

HRW18, BEK+18b, BEK+19, Sch12] and can, thus, easily be reused. We discuss the roles in software development using domain specific languages already in [KRV06] and elaborate on the engineering aspect of DSL development in [CFJ+16].
[Wei12, HRW15, Hoe18] present an approach that allows the creation of transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11, GMR+16]. [BDL+18] presents a method to derive internal DSLs from grammars. In [BJRW18], we discuss the translation from grammars to accurate metamodels. Successful applications have been carried out in the Air Traffic Management [ZPK+11] and television [DHH+20] domains. Based on the concepts described above, meta modeling, model analyses, and model evolution have been discussed in [LRSS10] and [SRVK10]. [BJRW18] describes a mapping bridge between both. DSL quality in [FHR08], instructions for defining views [GHK+07] and [PFR02], guidelines to define DSLs [KKP+09], and Eclipse-based tooling for DSLs [KRV07a] complete the collection. A broader discussion on the *global* integration of DSMLs is given in [CBCR15] as part of [CCF+15a], and [TAB+21] discusses the compositionality of analysis techniques for models.
The MontiCore language workbench has been successfully applied to a larger number of domains, resulting in a variety of languages documented, e.g., in [AHRW17, BEH+20, BHR+21, BPR+20, HHR+15, HJRW20, HMR+19, HRR12, PBI+16, RRW15] and Ph.D. theses like [Ber10, Gre19, Hab16, Her19, Kus21, Loo17, Pin14, Plo18, Rei16, Rot17, Sch12, Wor16].

Software Language Engineering

For a systematic definition of languages using a composition of reusable and adaptable language components, we adopt an engineering viewpoint on these techniques. General ideas on how to engineer a language can be found in the GeMoC initiative [CBCR15, CCF+15a]. As said, the MontiCore language workbench provides techniques for an integrated definition of languages [KRV07b, Kra10, KRV10, HR17, HKR21, HRW18, BPR+20, BEK+19].
In [SRVK10] we discuss the possibilities and the challenges of using metamodels for language definition. Modular composition, however, is a core concept to reuse language components like in MontiCore for the frontend [Voe11, Naz17, KRV08, HLN+15, HLN+15a, HNRW16, HKR21, BEK+18b, BEK+19] and the backend [RRRW15b, NRR16, GMR+16, HKR21, BEK+18b, BBC+18]. In [GHK+15, GHK+15a], we discuss the integration of handwritten and generated object-oriented code. [KRV10] describes the roles in software development using domain specific languages.
Language derivation is to our belief a promising technique to develop new languages for a specific purpose, e.g., model transformation, that relies on existing basic languages [HRW18].
How to automatically derive such a transformation language using a concrete syntax of the base language is described in [HRW15, Wei12] and successfully applied to various DSLs.
We also applied the language derivation technique to tagging languages that decorate a base language [GLRR15] and delta languages [HHK+15, HHK+13] that are derived from base languages to be able to constructively describe differences between model variants usable to build feature sets.
The derivation of internal DSLs from grammars is discussed in [BDL+18] and a translation of grammars to accurate metamodels in [BJRW18].

Modeling Software Architecture & the MontiArc Tool

Distributed interactive systems communicate via messages on a bus, discrete event signals, streams of telephone or video data, method invocation, or data structures passed between software services.
We use streams, statemachines, and components [BR07] as well as expressive forms of composition and refinement [PR99, RW18] for semantics. Furthermore, we built a concrete tooling infrastructure called MontiArc [HRR10, HRR12] for architecture design and extensions for states [RRW13c, BKRW17a, RRW14a, Wor16]. In [RRW13], we introduce a code generation framework for MontiArc. [RRRW15b] describes how the language is composed of individual sublanguages.
MontiArc was extended to describe variability [HRR+11] using deltas [HRRS11, HKR+11] and evolution on deltas [HRRS12]. Other extensions are concerned with modeling cloud architectures [PR13], security in [HHR+15], and the robotics domain [AHRW17, AHRW17b]. Extension mechanisms for MontiArc are generally discussed in [BHH+17].
[GHK+07] and [GHK+08] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants.
[MRR14b] provides a precise technique for verifying the consistency of architectural views [Rin14, MRR13] against a complete architecture to increase reusability. We discuss the synthesis problem for these views in [MRR14a]. An experience report [MRRW16] and a methodological embedding [DGH+19] complete the core approach.
Extensions for co-evolution of architecture are discussed in [MMR10], for powerful analyses of software architecture behavior evolution provided in [BKRW19], techniques for understanding semantic differences presented in [BKRW17], and modeling techniques to describe dynamic architectures shown in [HRR98, HKR+16, BHK+17, KKR19].

Compositionality & Modularity of Models

[HKR+09, TAB+21] motivate the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07, RW18] and algebraically grounded in [HKR+07]. Semantic and methodical aspects of model composition [KRV08] led to the language workbench MontiCore [KRV10, HKR21] that can even be used to develop modeling tools in a compositional form [HKR21, HLN+15, HLN+15a, HNRW16, NRR16, HRW18, BEK+18b, BEK+19, BPR+20, KRV07b]. A set of DSL design guidelines incorporates reuse through this form of composition [KKP+09].
[Voe11] examines the composition of context conditions respectively the underlying infrastructure of the symbol table. Modular editor generation is discussed in [KRV07a]. [RRRW15b] applies compositionality to robotics control.
[CBCR15] (published in [CCF+15a]) summarizes our approach to composition and remaining challenges in form of a conceptual model of the "globalized" use of DSLs. As a new form of decomposition of model information, we have developed the concept of tagging languages in [GLRR15, MRRW16]. It allows the description of additional information for model elements in separated documents, facilitates reuse, and allows typing tags.

Semantics of Modeling Languages

The meaning of semantics and its principles like underspecification, language precision, and detailedness is discussed in [HR04]. We defined a semantic domain called "System Model" by

using mathematical theory in [RKB95, BHP+98] and [GKR96, KRB96, RK96]. An extended version especially suited for the UML is given in [GRR09], [BCGR09a] and in [BCGR09] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied to class diagrams in [CGR08] or sequence diagrams in [BGH+98a].
To better understand the effect of an evolved design, detection of semantic differencing, as opposed to pure syntactical differences, is needed [MRR10]. [MRR11d, MRR11a] encode a part of the semantics to handle semantic differences of activity diagrams. [MRR11f, MRR11f] compare class and object diagrams with regard to their semantics. And [BKRW17] compares component and connector architectures similar to SysML' block definition diagrams.
In [BR07, RR11], a precise mathematical model for distributed systems based on black-box behaviors of components is defined and accompanied by automata in [Rum96]. Meta-modeling semantics is discussed in [EFLR99]. [BGH+97] discusses potential modeling languages for the description of exemplary object interaction, today called sequence diagram. [BGH+98b] discusses the relationships between a system, a view, and a complete model in the context of the UML.
[GR11] and [CGR09] discuss general requirements for a framework to describe semantic and syntactic variations of a modeling language. We apply these to class and object diagrams in [MRR11f] as well as activity diagrams in [GRR10].
[Rum12] defines the semantics in a variety of code and test case generation, refactoring, and evolution techniques. [LRSS10] discusses the evolution and related issues in greater detail. [RW18] discusses an elaborated theory for the modeling of underspecification, hierarchical composition, and refinement that can be practically applied to the development of CPS.
A first encoding of these theories in the Isabelle verification tool is defined in [BKR+20].

Evolution and Transformation of Models

Models are the central artifacts in model driven development, but as code, they are not initially correct and need to be changed, evolved, and maintained over time. Model transformation is therefore essential to effectively deal with models [CFJ+16].
Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04c, MRR10], refinement [PR99, KPR97, PR94], decomposition [PR99, KRW20], synthesis [MRR14a], refactoring [Rum12, PR03], translating models from one language into another [MRR11e, Rum12], systematic model transformation language development [Wei12, HRW15, Hoe18, HHR+15], repair of failed model evolution [KR18a].
[Rum04c] describes how comprehensible sets of such transformations support software development and maintenance [LRSS10], technologies for evolving models within a language and across languages, and mapping architecture descriptions to their implementation [MMR10]. Automaton refinement is discussed in [PR94, KPR97] and refining pipe-and-filter architectures is explained in [PR99]. This has e.g. been applied for robotics in [AHRW17, AHRW17b].
Refactorings of models are important for model driven engineering as discussed in [PR01, PR03, Rum12]. [HRRS11, HRR+11, HRRS12] encode these in constructive Delta transformations, which are defined in derivable Delta languages [HHK+13].
Translation between languages, e.g., from class diagrams into Alloy [MRR11e] allows for comparing class diagrams on a semantic level. Similarly, semantic differences of evolved activity diagrams are identified via techniques from [MRR11d] and for Simulink models in [RSW+15].

Variability and Software Product Lines (SPL)

Products often exist in various variants, for example, cars or mobile phones, where one manufacturer develops several products with many similarities but also many variations. Variants are managed in a Software Product Line (SPL) that captures product commonalities as well as differences. Feature diagrams describe variability in a top down fashion, e.g., in the automotive domain [GHK+08, GKPR08] using 150% models. Reducing overhead and associated costs is discussed in [GRJA12].

Delta modeling is a bottom up technique starting with a small, but complete base variant. Features are additive, but also can modify the core. A set of commonly applicable deltas configures a system variant. We discuss the application of this technique to Delta-MontiArc [HRRS11, HRR+11] and to Delta-Simulink [HKM+13]. Deltas can not only describe special variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK+13, HHK+15] and [HRW15] describe an approach to systematically derive delta languages.

We also apply variability modeling languages to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02] and generators [GMR+16]. Furthermore, we specified a systematic way to define variants of modeling languages [CGR09], leverage features for their compositional reuse [BEK+18b, BEK+19], and applied it as a semantic language refinement on Statecharts in [GR11].

Digital Twins and Digital Shadows in Engineering and Production

The digital transformation of production changes the life cycle of the design, the production, and the use of products [BDJ+22]. To support this transformation, we can use Digital Twins (DTs) and Digital Shadows (DSs). In [DMR+20] we define: "A digital twin of a system consists of a set of models of the system, a set of digital shadows, and provides a set of services to use the data and models purposefully with respect to the original system."

We have investigated how to synthesize self-adaptive DT architectures with model-driven methods [BBD+21a] and have applied it e.g. on a digital twin for injection molding [BDH+20]. In [BDR+21] we investigate the economic implications of digital twin services.

Digital twins also need user interaction and visualization, why we have extended the infrastructure by generating DT cockpits [DMR+20]. To support the DevOps approach in DT engineering, we have created a generator for low-code development platforms for digital twins [DHM+22] and sophisticated tool chains to generate process-aware digital twin cockpits that also include condensed forms of event logs [BMR+22].

[BBD+21b] describes a conceptual model for digital shadows covering the purpose, relevant assets, data, and metadata as well as connections to engineering models. These can be used during the runtime of a DT, e.g. when using process prediction services within DTs [BHK+21]. Integration challenges for digital twin systems-of-systems [MPRW22] include, e.g., the horizontal integration of digital twin parts, the composition of DTs for different perspectives, or how to handle different lifecycle representations of the original system.

Modeling for Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12, BBR20] are complex, distributed systems that control physical entities. In [RW18], we discuss how an elaborated theory can be practically applied to

the development of CPS. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12, KRRW17], autonomous driving [BR12b, KKR19], and digital twin development [BDH+20] to processes and tools to improve the development as well as the product itself [BBR07].
In the aviation domain, a modeling language for uncertainty and safety events was developed, which is of interest to European avionics [ZPK+11]. Optimized [KRS+18a] and domain specific code generation [AHRW17b], and the extension to product lines of CPS [RSW+15, KRR+16, RRS+16] are key for CPS.
A component and connector architecture description language (ADL) suitable for the specific challenges in robotics is discussed in [RRW13c, RRW14a, Wor16, RRSW17, Wor21]. In [RRW12], we use this language for describing requirements and in [RRW13], we describe a code generation framework for this language. Monitoring for smart and energy efficient buildings is developed as an Energy Navigator toolset [KPR12, FPPR12, KLPR12].

Model-Driven Systems Engineering (MDSysE)

Applying models during Systems Engineering activities is based on the long tradition of contributing to systems engineering in automotive [FND+98] and [GHK+08a], which culminated in a new comprehensive model-driven development process for automotive software [KMS+18, DGH+19]. We leveraged SysML to enable the integrated flow from requirements to implementation to integration.
To facilitate the modeling of products, resources, and processes in the context of Industry 4.0, we also conceived a multi-level framework for production engineering based on these concepts [BKL+18] and addressed to bridge the gap between functions and the physical product architecture by modeling mechanical functional architectures in SysML [DRW+20]. For that purpose, we also did a detailed examination of the upcoming SysML 2.0 standard [JPR+22] and examined how to extend the SPES/CrEST methodology for a systems engineering approach [BBR20].
Research within the excellence cluster Internet of Production considers fast decision making at production time with low latencies using contextual data traces of production systems, also known as Digital Shadows (DS) [SHH+20]. We have investigated how to derive Digital Twins (DTs) for injection molding [BDH+20], how to generate interfaces between a cyber-physical system and its DT [KMR+20], and have proposed model-driven architectures for DT cockpit engineering [DMR+20].

State Based Modeling (Automata)

Today, many computer science theories are based on statemachines in various forms including Petri nets or temporal logics. Software engineering is particularly interested in using statemachines for modeling systems. Our contributions to state based modeling can currently be split into three parts: (1) understanding how to model object-oriented and distributed software using statemachines resp. Statecharts [GKR96, BCR07b, BCGR09a, BCGR09], (2) understanding the refinement [PR94, RK96, Rum96, RW18] and composition [GR95, GKR96, RW18] of statemachines, and (3) applying statemachines for modeling systems.
In [Rum96, RW18] constructive transformation rules for refining automata behavior are given

and proven correct. This theory is applied to features in [KPR97]. Statemachines are embedded in the composition and behavioral specification concepts of Focus [GKR96, BR07].
We apply these techniques, e.g., in MontiArcAutomaton [RRW13, RRW14a, RRW13, RW18], in a robot task modeling language [THR+13], and in building management systems [FLP+11b].

Model-Based Assistance and Information Services (MBAIS)

Assistive systems are a special type of information system: they (1) provide situational support for human behavior (2) based on information from previously stored and real-time monitored structural context and behavior data (3) at the time the person needs or asks for it [HMR+19]. To create them, we follow a model centered architecture approach [MMR+17] which defines systems as a compound of various connected models. Used languages for their definition include DSLs for behavior and structure such as the human cognitive modeling language [MM13], goal modeling languages [MRV20, MRZ21] or UML/P based languages [MNRV19]. [MM15] describes a process of how languages for assistive systems can be created. MontiGem [AMN+20a] is used as the underlying generator technology.
We have designed a system included in a sensor floor able to monitor elderlies and analyze impact patterns for emergency events [LMK+11]. We have investigated the modeling of human contexts for the active assisted living and smart home domain [MS17] and user-centered privacy-driven systems in the IoT domain in combination with process mining systems [MKM+19], differential privacy on event logs of handling and treatment of patients at a hospital [MKB+19], the mark-up of online manuals for devices [SM18a] and websites [SM20], and solutions for privacy-aware environments for cloud services [ELR+17] and in IoT manufacturing [MNRV19]. The user-centered view of the system design allows to track who does what, when, why, where, and how with personal data, makes information about it available via information services and provides support using assistive services.

Modeling Robotics Architectures and Tasks

Robotics can be considered a special field within Cyber-Physical Systems which is defined by an inherent heterogeneity of involved domains, relevant platforms, and challenges. The engineering of robotics applications requires the composition and the interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers the broad propagation of robotics applications.
The MontiArcAutomaton language [RRW12, RRW14a] extends the ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13c, RRRW15b, HKR21] that perfectly fit robotic architectural modeling.
The iserveU modeling framework describes domains, actors, goals, and tasks of service robotics applications [ABH+16, ABH+17] with declarative models. Goals and tasks are translated into models of the planning domain definition language (PDDL) and then solved [ABK+17]. Thus, domain experts focus on describing the domain and its properties only.
The LightRocks [THR+13, BRS+15] framework allows robotics experts and laymen to model robotic assembly tasks. In [AHRW17, AHRW17b], we define a modular architecture modeling method for translating architecture models into modules compatible with different robotics middleware platforms.

Many of the concepts in robotics were derived from automotive software [BBR07, BR12b].

Automotive, Autonomic Driving & Intelligent Driver Assistance

Introducing and connecting sophisticated driver assistance, infotainment, and communication systems as well as advanced active and passive safety-systems result in complex embedded systems. As these feature-driven subsystems may be arbitrarily combined by the customer, a huge amount of distinct variants needs to be managed, developed, and tested. A consistent requirement management connecting requirements with features in all development phases for the automotive domain is described in [GRJA12].
The conceptual gap between requirements and the logical architecture of a car is closed in [GHK+07, GHK+08]. A methodical embedding of the resulting function nets and their quality assurance using automated testing is given in the SMaRDT method [DGH+19, KMS+18].
[HKM+13] describes a tool for delta modeling for Simulink [HKM+13]. [HRRW12] discusses the means to extract a well-defined Software Product Line from a set of copy and paste variants.
Potential variants of components in product lines can be identified using similarity analysis of interfaces [KRR+16], or execute tests to identify similar behavior [RRS+16]. [RSW+15] describes an approach to using model checking techniques to identify behavioral differences of Simulink models. In [KKR19], we model dynamic reconfiguration of architectures applied to cooperating vehicles.
Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12b, BR12], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in the development and the evolution of autonomous car functionality, and thus enables us to develop software in an agile way [BR12b].
[MMR10] gives an overview of the state-of-the-art in development and evolution on a more general level by considering any kind of critical system that relies on architectural descriptions.
MontiSim simulates autonomous and cooperative driving behavior [GKR+17] for testing various forms of errors as well as spatial distance [FIK+18, KKRZ19]. As tooling infrastructure, the SSELab storage, versioning, and management services [HKR12] are essential for many projects.

Internet of Things, Industry 4.0 & the MontiThings Tool

The Internet of Things (IoT) requires the development of increasingly complex distributed systems. The MontiThings ecosystem [KRS+22] provides an end-to-end solution to modeling, deploying [KKR+22], and analyzing [KMR21] failure-tolerant [KRS+22] IoT systems and connecting them to synthesized digital twins [KMR+20]. We have investigated how model-driven methods can support the development of privacy-aware [ELR+17, HHK+14] cloud systems [PR13], distributed systems security [HHR+15], privacy-aware process mining [MKM+19], and distributed robotics applications [RRRW15b].
In the course of Industry 4.0, we have also turned our attention to mechanical or electrical applications [DRW+20]. We identified the digital representation, integration, and (re-)configuration of automation systems as primary Industry 4.0 concerns [WCB17]. Using a multi-level modeling framework, we support machine as a service approaches [BKL+18].

Smart Energy Management

In the past years, it became more and more evident that saving energy and reducing CO_2 emissions are important challenges. Thus, energy management in buildings as well as in neighborhoods becomes equally important to efficiently use the generated energy. Within several research projects, we developed methodologies and solutions for integrating heterogeneous systems at different scales.
During the design phase, the Energy Navigators Active Functional Specification (AFS) [FPPR12, KPR12] is used for the technical specification of building services already.
We adapted the well-known concept of statemachines to be able to describe different states of a facility and validate it against the monitored values [FLP+11b]. We show how our data model, the constraint rules, and the evaluation approach to compare sensor data can be applied [KLPR12].

Cloud Computing and Services

The paradigm of Cloud Computing is arising out of a convergence of existing technologies for web-based application and service architectures with high complexity, criticality, and new application domains. It promises to enable new business models, facilitate web-based innovations, and increase the efficiency and cost-effectiveness of web development [KRR14].
Application classes like Cyber-Physical Systems and their privacy [HHK+14, HHK+15a], Big Data, Apps, and Service Ecosystems bring attention to aspects like responsiveness, privacy, and open platforms. Regardless of the application domain, developers of such systems need robust methods and efficient, easy-to-use languages and tools [KRS12].
We tackle these challenges by perusing a model-based, generative approach [PR13]. At the core of this approach are different modeling languages that describe different aspects of a cloud-based system in a concise and technology-agnostic way. Software architecture and infrastructure models describe the system and its physical distribution on a large scale.
We apply cloud technology for the services we develop, e.g., the SSELab [HKR12] and the Energy Navigator [FPPR12, KPR12] but also for our tool demonstrators and our development platforms. New services, e.g., for collecting data from temperature sensors, cars, etc. are now easily developed and deployed, e.g., in production or Internet-of-Things environments.
Security aspects and architectures of cloud services for the digital me in a privacy-aware environment are addressed in [ELR+17].

Model-Driven Engineering of Information Systems & the MontiGem Tool

Information Systems provide information to different user groups as the main system goal. Using our experiences in the model-based generation of code with MontiCore [KRV10, HKR21], we developed several generators for such data-centric information systems.
MontiGem [AMN+20a] is a specific generator framework for data-centric business applications that uses standard models from UML/P optionally extended by GUI description models as sources [GMN+20]. While the standard semantics of these modeling languages remains untouched, the generator produces a lot of additional functionality around these models. The generator is designed flexible, modular, and incremental, handwritten and generated code pieces are well integrated [GHK+15a, NRR15a], tagging of existing models is possible [GLRR15], e.g., for the definition of roles and rights or for testing [DGH+18].

We are using MontiGem for financial management [GHK+20, ANV+18], for creating digital twin cockpits [DMR+20], and various industrial projects. MontiGem makes it easier to create low-code development platforms for digital twins [DHM+22]. When using additional DSLs, we can develop assistive systems providing user support based on goal models [MRV20], privacy-preserving information systems using privacy models and purpose trees [MNRV19], and process-aware digital twin cockpits using BPMN models [BMR+22].
We have also developed an architecture of cloud services for the digital me in a privacy-aware environment [ELR+17] and a method for retrofitting generative aspects into existing applications [DGM+21].

[ABH+16] Kai Adam, Arvid Butting, Robert Heim, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Model-Driven Separation of Concerns for Service Robotics. In *International Workshop on Domain-Specific Modeling (DSM'16)*, pages 22–27. ACM, October 2016.

[ABH+17] Kai Adam, Arvid Butting, Robert Heim, Oliver Kautz, Jérôme Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. *Modeling Robotics Tasks for Better Separation of Concerns, Platform-Independence, and Reuse.* Aachener Informatik-Berichte, Software Engineering, Band 28. Shaker Verlag, December 2017.

[ABK+17] Kai Adam, Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Executing Robot Task Models in Dynamic Environments. In *Proceedings of MODELS 2017. Workshop EXE*, CEUR 2019, September 2017.

[AHRW17] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Engineering Robotics Software Architectures with Exchangeable Model Transformations. In *International Conference on Robotic Computing (IRC'17)*, pages 172–179. IEEE, April 2017.

[AHRW17b] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Modeling Robotics Software Architectures with Modular Model Transformations. *Journal of Software Engineering for Robotics (JOSER)*, 8(1):3–16, 2017.

[AKK+21] Abdallah Atouani, Jörg Christian Kirchhof, Evgeny Kusmenko, and Bernhard Rumpe. Artifact and Reference Models for Generative Machine Learning Frameworks and Build Systems. In Eli Tilevich and Coen De Roover, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 21)*, pages 55–68. ACM, October 2021.

[AMN+20a] Kai Adam, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Enterprise Information Systems in Academia and Practice: Lessons learned from a MBSE Project. In *40 Years EMISA: Digital Ecosystems of the Future: Methodology, Techniques and Applications (EMISA'19)*, LNI P-304, pages 59–66. Gesellschaft für Informatik e.V., May 2020.

[ANV+18] Kai Adam, Lukas Netz, Simon Varga, Judith Michael, Bernhard Rumpe, Patricia Heuser, and Peter Letmathe. Model-Based Generation of Enterprise Information Systems. In Michael Fellmann and Kurt Sandkuhl, editors, *Enterprise Modeling and Information Systems Architectures (EMISA'18)*, CEUR Workshop Proceedings 2097, pages 75–79. CEUR-WS.org, May 2018.

[BBC+18] Inga Blundell, Romain Brette, Thomas A. Cleland, Thomas G. Close, Daniel Coca, Andrew P. Davison, Sandra Diaz-Pier, Carlos Fernandez Musoles, Padraig Gleeson, Dan F. M. Goodman, Michael Hines, Michael W. Hopkins, Pramod Kumbhar, David R. Lester, Bóris Marin, Abigail Morrison, Eric Müller, Thomas Nowotny, Alexander Peyser, Dimitri Plotnikov, Paul Richmond, Andrew Rowley, Bernhard Rumpe, Marcel Stimberg, Alan B. Stokes, Adam Tomkins, Guido Trensch, Marmaduke Woodman, and Jochen Martin Eppler. Code Generation in Computational Neuroscience: A Review of Tools and Techniques. *Journal Frontiers in Neuroinformatics*, 12, 2018.

[BBD+21b] Fabian Becker, Pascal Bibow, Manuela Dalibor, Aymen Gannouni, Viviane Hahn, Christian Hopmann, Matthias Jarke, Istvan Koren, Moritz Kröger, Johannes Lipp, Judith Maibaum, Judith Michael, Bernhard Rumpe, Patrick Sapel, Niklas Schäfer, Georg J. Schmitz, Günther Schuh, and Andreas Wortmann. A Conceptual Model for Digital Shadows in Industry and its Application. In Aditya Ghose, Jennifer Horkoff, Vitor E. Silva Souza, Jeffrey Parsons, and Joerg Evermann, editors, *Conceptual Modeling, ER 2021*, pages 271–281. Springer, October 2021.

[BBD+21a] Tim Bolender, Gereon Bürvenich, Manuela Dalibor, Bernhard Rumpe, and Andreas Wortmann. Self-Adaptive Manufacturing with Digital Twins. In *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 156–166. IEEE Computer Society, May 2021.

[BBR07] Christian Basarke, Christian Berger, and Bernhard Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 4(12):1158–1174, 2007.

[BBR20] Manfred Broy, Wolfgang Böhm, and Bernhard Rumpe. Advanced Systems Engineering - Die Systeme der Zukunft. White paper, fortiss. Forschungsinstitut für softwareintensive Systeme, Munich, July 2020.

[BCGR09] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, November 2009.

[BCGR09a] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, November 2009.

[BCR07a] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, Germany, February 2007.

[BCR07b] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, Germany, February 2007.

[BDH+20] Pascal Bibow, Manuela Dalibor, Christian Hopmann, Ben Mainz, Bernhard Rumpe, David Schmalzing, Mauritius Schmitz, and Andreas Wortmann. Model-Driven Development of a Digital Twin for Injection Molding. In Schahram Dustdar, Eric Yu, Camille Salinesi, Dominique Rieu, and Vik Pant, editors, *International Conference on Advanced Information Systems Engineering (CAiSE'20)*, Lecture Notes in Computer Science 12127, pages 85–100. Springer International Publishing, June 2020.

[BDJ+22] Philipp Brauner, Manuela Dalibor, Matthias Jarke, Ike Kunze, István Koren, Gerhard Lakemeyer, Martin Liebenberg, Judith Michael, Jan Pennekamp, Christoph Quix, Bernhard Rumpe, Wil van der Aalst, Klaus Wehrle, Andreas

Wortmann, and Martina Ziefle. A Computer Science Perspective on Digital Transformation in Production. *Journal ACM Transactions on Internet of Things*, 3:1–32, February 2022.

[BDL+18] Arvid Butting, Manuela Dalibor, Gerrit Leonhardt, Bernhard Rumpe, and Andreas Wortmann. Deriving Fluent Internal Domain-specific Languages from Grammars. In *International Conference on Software Language Engineering (SLE'18)*, pages 187–199. ACM, 2018.

[BDR+21] Christian Brecher, Manuela Dalibor, Bernhard Rumpe, Katrin Schilling, and Andreas Wortmann. An Ecosystem for Digital Shadows in Manufacturing. In *54th CIRP CMS 2021 - Towards Digitalized Manufacturing 4.0*. Elsevier, September 2021.

[BEH+20] Arvid Butting, Robert Eikermann, Katrin Hölldobler, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. A Library of Literals, Expressions, Types, and Statements for Compositional Language Design. *Journal of Object Technology (JOT)*, 19(3):3:1–16, October 2020.

[BEK+18b] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Modeling Language Variability with Reusable Language Components. In *International Conference on Systems and Software Product Line (SPLC'18)*. ACM, September 2018.

[BEK+19] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Systematic Composition of Independent Language Features. *Journal of Systems and Software (JSS)*, 152:50–69, June 2019.

[Ber10] Christian Berger. *Automating Acceptance Tests for Sensor- and Actuator-based Systems on the Example of Autonomous Vehicles*. Aachener Informatik-Berichte, Software Engineering, Band 6. Shaker Verlag, 2010.

[BGH+97] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Towards a Precise Semantics for Object-Oriented Modeling Techniques. In Jan Bosch and Stuart Mitchell, editors, *Object-Oriented Technology, ECOOP'97 Workshop Reader*, LNCS 1357. Springer Verlag, 1997.

[BGH+98a] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. *Journal Computer Standards & Interfaces*, 19(7):335–345, November 1998.

[BGH+98b] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, Germany, 1998.

[BGRW17] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. Taming the Complexity of Model-Driven Systems Engineering Projects. In *Part of the Grand Challenges in Modeling (GRAND'17) Workshop*, July 2017.

[BGRW18] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. On the Need for Artifact Models in Model-Driven Systems Engineering Projects. In Martina Seidl and Steffen Zschaler, editors, *Software Technologies: Applications and Foundations*, LNCS 10748, pages 146–153. Springer, January 2018.

[BHH+17] Arvid Butting, Arne Haber, Lars Hermerschmidt, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Systematic Language Extension Mechanisms for the MontiArc Architecture Description Language. In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 53–70. Springer, July 2017.

[BHK+17] Arvid Butting, Robert Heim, Oliver Kautz, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Classification of Dynamic Reconfiguration in Component and Connector Architecture Description Languages. In *Proceedings of MODELS 2017. Workshop ModComp*, CEUR 2019, September 2017.

[BHK+21] Tobias Brockhoff, Malte Heithoff, István Koren, Judith Michael, Jérôme Pfeiffer, Bernhard Rumpe, Merih Seran Uysal, Wil M. P. van der Aalst, and Andreas Wortmann. Process Prediction with Digital Twins. In *Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 182–187. ACM/IEEE, October 2021.

[BHP+98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In *Workshop on Requirements Targeting Software and Systems Engineering (RTSE'97)*, LNCS 1526, pages 43–68. Springer, 1998.

[BHR+18] Arvid Butting, Steffen Hillemacher, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. Shepherding Model Evolution in Model-Driven Development. In *Joint Proceedings of the Workshops at Modellierung 2018 (MOD-WS 2018)*, CEUR Workshop Proceedings 2060, pages 67–77. CEUR-WS.org, February 2018.

[BHR+21] Arvid Butting, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Compositional Modelling Languages with Analytics and Construction Infrastructures Based on Object-Oriented Techniques - The MontiCore Approach. In Heinrich, Robert and Duran, Francisco and Talcott, Carolyn and Zschaler, Steffen, editor, *Composing Model-Based Analysis Tools*, pages 217–234. Springer, July 2021.

[BJRW18] Arvid Butting, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. Translating Grammars to Accurate Metamodels. In *International Conference on Software Language Engineering (SLE'18)*, pages 174–186. ACM, 2018.

[BKL+18] Christian Brecher, Evgeny Kusmenko, Achim Lindt, Bernhard Rumpe, Simon Storms, Stephan Wein, Michael von Wenckstern, and Andreas Wortmann. Multi-Level Modeling Framework for Machine as a Service Applications Based on Product Process Resource Models. In *Proceedings of the 2nd International Symposium on Computer Science and Intelligent Control (ISCSIC'18)*. ACM, September 2018.

[BKR+20] Jens Christoph Bürger, Hendrik Kausch, Deni Raco, Jan Oliver Ringert, Bernhard Rumpe, Sebastian Stüber, and Marc Wiartalla. *Towards an Isabelle Theory for distributed, interactive systems - the untimed case.* Aachener Informatik Berichte, Software Engineering, Band 45. Shaker Verlag, March 2020.

[BKRW17a] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Architectural Programming with MontiArcAutomaton. In *In 12th International Conference on Software Engineering Advances (ICSEA 2017)*, pages 213–218. IARIA XPS Press, May 2017.

[BKRW17] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Semantic Differencing for Message-Driven Component & Connector Architectures. In *International Conference on Software Architecture (ICSA'17)*, pages 145–154. IEEE, April 2017.

[BKRW19] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Continuously Analyzing Finite, Message-Driven, Time-Synchronous Component & Connector Systems During Architecture Evolution. *Journal of Systems and Software (JSS)*, 149:437–461, March 2019.

[BMR+22] Dorina Bano, Judith Michael, Bernhard Rumpe, Simon Varga, and Matthias Weske. Process-Aware Digital Twin Cockpit Synthesis from Event Logs. *Journal of Computer Languages (COLA)*, 70, June 2022.

[BPR+20] Arvid Butting, Jerome Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. A Compositional Framework for Systematic Modeling Language Reuse. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 35–46. ACM, October 2020.

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007.

[BR12b] Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Automotive Software Engineering Workshop (ASE'12)*, pages 789–798, 2012.

[BR12] Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*, pages 243–271. Springer, Germany, 2012.

[BRS+15] Arvid Butting, Bernhard Rumpe, Christoph Schulze, Ulrike Thomas, and Andreas Wortmann. Modeling Reusable, Platform-Independent Robot Assembly Processes. In *International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2015)*, 2015.

[CBCR15] Tony Clark, Mark van den Brand, Benoit Combemale, and Bernhard Rumpe. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*, LNCS 9400, pages 7–20. Springer, 2015.

[CCF+15a] Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe, editors. *Globalizing Domain-Specific Languages*, LNCS 9400. Springer, 2015.

[CEG+14] Betty H.C. Cheng, Kerstin I. Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi A. Müller, Patrizio Pelliccione, Anna Perini, Nauman A. Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha M. Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In Nelly Bencomo, Robert France, Betty H.C. Cheng, and Uwe Aßmann, editors, *Models@run.time*, LNCS 8378, pages 101–136. Springer International Publishing, Switzerland, 2014.

[CFJ+16] Benoit Combemale, Robert France, Jean-Marc Jézéquel, Bernhard Rumpe, James Steel, and Didier Vojtisek. *Engineering Modeling Languages: Turning Domain Knowledge into Tools*. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series, November 2016.

[CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany, 2008.

[CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, LNCS 5795, pages 670–684. Springer, 2009.

[DEKR19] Imke Drave, Robert Eikermann, Oliver Kautz, and Bernhard Rumpe. Semantic Differencing of Statecharts for Object-oriented Systems. In Slimane Hammoudi, Luis Ferreira Pires, and Bran Selić, editors, *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'19)*, pages 274–282. SciTePress, February 2019.

[DGH+18] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Matthias Markthaler, Bernhard Rumpe, and Andreas Wortmann. Model-Based Testing of Software-Based System Functions. In *Conference on Software Engineering and Advanced Applications (SEAA'18)*, pages 146–153, August 2018.

[DGH+19] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Evgeny Kusmenko, Matthias Markthaler, Philipp Orth, Karin Samira Salman, Johannes Richenhagen, Bernhard Rumpe, Christoph Schulze, Michael Wenckstern, and Andreas Wortmann. SMArDT modeling for automotive software testing. *Journal on Software: Practice and Experience*, 49(2):301–328, February 2019.

[DGM+21] Imke Drave, Akradii Gerasimov, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. A Methodology for Retrofitting Generative Aspects in Existing Applications. *Journal of Object Technology (JOT)*, 20:1–24, November 2021.

[DHH+20] Imke Drave, Timo Henrich, Katrin Hölldobler, Oliver Kautz, Judith Michael, and Bernhard Rumpe. Modellierung, Verifikation und Synthese von validen Planungszuständen für Fernsehausstrahlungen. In Dominik Bork, Dimitris Karagiannis, and Heinrich C. Mayr, editors, *Modellierung 2020*, pages 173–188. Gesellschaft für Informatik e.V., February 2020.

[DHM+22] Manuela Dalibor, Malte Heithoff, Judith Michael, Lukas Netz, Jérôme Pfeiffer, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Generating Customized Low-Code Development Platforms for Digital Twins. *Journal of Computer Languages (COLA)*, 70, June 2022.

[DKMR19] Imke Drave, Oliver Kautz, Judith Michael, and Bernhard Rumpe. Semantic Evolution Analysis of Feature Models. In Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi, editors, *International Systems and Software Product Line Conference (SPLC'19)*, pages 245–255. ACM, September 2019.

[DMR+20] Manuela Dalibor, Judith Michael, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Towards a Model-Driven Architecture for Interactive Digital Twin Cockpits. In Gillian Dobbie, Ulrich Frank, Gerti Kappel, Stephen W. Liddle, and Heinrich C. Mayr, editors, *Conceptual Modeling*, pages 377–387. Springer International Publishing, October 2020.

[DRW+20] Imke Drave, Bernhard Rumpe, Andreas Wortmann, Joerg Berroth, Gregor Hoepfner, Georg Jacobs, Kathrin Spuetz, Thilo Zerwas, Christian Guist, and Jens Kohl. Modeling Mechanical Functional Architectures in SysML. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 79–89. ACM, October 2020.

[EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Meta-Modelling Semantics of UML. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 45–60. Kluver Academic Publisher, 1999.

[EFLR99a] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. The UML as a Formal Modeling Notation. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language. «UML»'98: Beyond the Notation*, LNCS 1618, pages 336–348. Springer, Germany, 1999.

[EJK+19] Rolf Ebert, Jahir Jolianis, Stefan Kriebel, Matthias Markthaler, Benjamin Pruenster, Bernhard Rumpe, and Karin Samira Salman. Applying Product Line Testing for the Electric Drive System. In Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi, editors, *International Systems and Software Product Line Conference (SPLC'19)*, pages 14–24. ACM, September 2019.

[ELR+17] Robert Eikermann, Markus Look, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Architecting Cloud Services for the Digital me in a Privacy-Aware Environment. In Ivan Mistrik, Rami Bahsoon, Nour Ali, Maritta Heisel, and Bruce Maxim, editors, *Software Architecture for Big Data and the Cloud*, chapter 12, pages 207–226. Elsevier Science & Technology, June 2017.

[FEL+98] Robert France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. *Journal Computer Standards & Interfaces*, 19(7):325–334, November 1998.

[FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008.

[FIK+18] Christian Frohn, Petyo Ilov, Stefan Kriebel, Evgeny Kusmenko, Bernhard Rumpe, and Alexander Ryndin. Distributed Simulation of Cooperatively Interacting Vehicles. In *International Conference on Intelligent Transportation Systems (ITSC'18)*, pages 596–601. IEEE, 2018.

[FLP+11] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. Der Energie-Navigator - Performance-Controlling für Gebäude und Anlagen. *Technik am Bau (TAB) - Fachzeitschrift für Technische Gebäudeausrüstung*, Seiten 36-41, März 2011.

[FLP+11b] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. State-based Modeling of Buildings and Facilities. In *Enhanced Building Operations Conference (ICEBO'11)*, 2011.

[FND+98] Max Fuchs, Dieter Nazareth, Dirk Daniel, and Bernhard Rumpe. BMW-ROOM An Object-Oriented Method for ASCET. In *SAE'98, Cobo Center (Detroit, Michigan, USA)*, Society of Automotive Engineers, 1998.

[FPPR12] M. Norbert Fisch, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Energy Efficiency in Commercial Buildings Conference (IEECB'12)*, 2012.

[GHK+07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based Modeling of Function Nets. In *Object-oriented Modelling of Embedded Real-Time Systems Workshop (OMER4'07)*, 2007.

[GHK+08] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, 2008.

[GHK+08a] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. View-Centric Modeling of Automotive Logical Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV*, Informatik Bericht 2008-02. TU Braunschweig, 2008.

[GHK+15] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, pages 74–85. SciTePress, 2015.

[GHK+15a] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk

Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. Integration of Handwritten and Generated Object-Oriented Code. In *Model-Driven Engineering and Software Development*, Communications in Computer and Information Science 580, pages 112–132. Springer, 2015.

[GHK+20] Arkadii Gerasimov, Patricia Heuser, Holger Ketteniß, Peter Letmathe, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Generated Enterprise Information Systems: MDSE for Maintainable Co-Development of Frontend and Backend. In Judith Michael and Dominik Bork, editors, *Companion Proceedings of Modellierung 2020 Short, Workshop and Tools & Demo Papers*, pages 22–30. CEUR Workshop Proceedings, February 2020.

[GHR17] Timo Greifenberg, Steffen Hillemacher, and Bernhard Rumpe. *Towards a Sustainable Artifact Model: Artifacts in Generator-Based Model-Driven Projects.* Aachener Informatik-Berichte, Software Engineering, Band 30. Shaker Verlag, December 2017.

[GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen*, Informatik Bericht 2008-01, pages 76–89. TU Braunschweig, 2008.

[GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TU Munich, Germany, July 1996.

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0: Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006.

[GKR+07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering, Nashville*, Informatik-Bericht 4/2007. Johannes-Gutenberg-Universität Mainz, 2007.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008.

[GKR+17] Filippo Grazioli, Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Simulation Framework for Executing Component and Connector Models of Self-Driving Vehicles. In *Proceedings of MODELS 2017. Workshop EXE*, CEUR 2019, September 2017.

[GLPR15] Timo Greifenberg, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Energieeffiziente Städte - Herausforderungen und Lösungen aus Sicht des Software Engineerings. In Linnhoff-Popien, Claudia and Zaddach, Michael and Grahl, Andreas, Editor, *Marktplätze im Umbruch: Digitale Strategien für Services im*

Mobilen Internet, Xpert.press, Kapitel 56, Seiten 511-520. Springer Berlin Heidelberg, April 2015.

[GLRR15] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. Engineering Tagging Languages for DSLs. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 34–43. ACM/IEEE, 2015.

[GMN+20] Arkadii Gerasimov, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Continuous Transition from Model-Driven Prototype to Full-Size Real-World Enterprise Information Systems. In Bonnie Anderson, Jason Thatcher, and Rayman Meservy, editors, *25th Americas Conference on Information Systems (AMCIS 2020)*, AIS Electronic Library (AISeL), pages 1–10. Association for Information Systems (AIS), August 2020.

[GMR+16] Timo Greifenberg, Klaus Müller, Alexander Roth, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. Modeling Variability in Template-based Code Generators for Product Line Engineering. In *Modellierung 2016 Conference*, LNI 254, pages 141–156. Bonner Köllen Verlag, March 2016.

[GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TU Munich, Germany, October 1995.

[GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems*, LNCS 6662, pages 17–32. Springer, 2011.

[Gre19] Timo Greifenberg. *Artefaktbasierte Analyse modellgetriebener Softwareentwicklungsprojekte*. Aachener Informatik-Berichte, Software Engineering, Band 42. Shaker Verlag, August 2019.

[GRJA12] Tim Gülke, Bernhard Rumpe, Martin Jansen, and Joachim Axmann. High-Level Requirements Management and Complexity Costs in Automotive Development Projects: A Problem Statement. In *Requirements Engineering: Foundation for Software Quality (REFSQ'12)*, 2012.

[GRR09] Hans Grönniger, Jan Oliver Ringert, and Bernhard Rumpe. System Model-based Definition of Modeling Language Semantics. In *Proc. of FMOODS/FORTE 2009, LNCS 5522*, Lisbon, Portugal, 2009.

[GRR10] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, LNCS 6394, pages 331–345. Springer, 2010.

[Hab16] Arne Haber. *MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems*. Aachener Informatik-Berichte, Software Engineering, Band 24. Shaker Verlag, September 2016.

[Her19] Lars Hermerschmidt. *Agile Modellgetriebene Entwicklung von Software Security & Privacy*. Aachener Informatik-Berichte, Software Engineering, Band 41. Shaker Verlag, June 2019.

[HHK+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Software Product Line Conference (SPLC'13)*, pages 22–31. ACM, 2013.

[HHK+14] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. User-driven Privacy Enforcement for Cloud-based Services in the Internet of Things. In *Conference on Future Internet of Things and Cloud (FiCloud'14)*. IEEE, 2014.

[HHK+15] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, Ina Schaefer, and Christoph Schulze. Systematic Synthesis of Delta Modeling Languages. *Journal on Software Tools for Technology Transfer (STTT)*, 17(5):601–626, October 2015.

[HHK+15a] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. A comprehensive approach to privacy in the cloud-based Internet of Things. *Journal Future Generation Computer Systems*, 56:701–718, 2015.

[HHR+15] Lars Hermerschmidt, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Generating Domain-Specific Transformation Languages for Component & Connector Architecture Descriptions. In *Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp'15)*, CEUR Workshop Proceedings 1463, pages 18–23, 2015.

[HJK+21] Steffen Hillemacher, Nicolas Jäckel, Christopher Kugler, Philipp Orth, David Schmalzing, and Louis Wachtmeister. Artifact-Based Analysis for the Development of Collaborative Embedded Systems. In *Model-Based Engineering of Collaborative Embedded Systems*, pages 315–331. Springer, January 2021.

[HJRW20] Katrin Hölldobler, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. Komposition Domänenspezifischer Sprachen unter Nutzung der MontiCore Language Workbench, am Beispiel SysML 2. In Dominik Bork, Dimitris Karagiannis, and Heinrich C. Mayr, editors, *Modellierung 2020*, pages 189–190. Gesellschaft für Informatik e.V., February 2020.

[HKM+13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-Class Variability Modeling in Matlab/Simulink. In *Variability Modelling of Software-intensive Systems Workshop (VaMoS'13)*, pages 11–18. ACM, 2013.

[HKR+07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In *Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'07)*, LNCS 4530, pages 99–113. Springer, Germany, 2007.

[HKR+09] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In *Conference on Software Engineeering in Research and Practice (SERP'09)*, pages 172–176, July 2009.

[HKR+11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *Software Architecture Conference (ECSA'11)*, pages 6:1–6:10. ACM, 2011.

[HKR12] Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Developing Tools as Plug-Ins Workshop (TOPI'12)*, pages 61–66. IEEE, 2012.

[HKR+16] Robert Heim, Oliver Kautz, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Retrofitting Controlled Dynamic Reconfiguration into the Architecture Description Language MontiArcAutomaton. In *Software Architecture - 10th European Conference (ECSA'16)*, LNCS 9839, pages 175–182. Springer, December 2016.

[HKR21] Katrin Hölldobler, Oliver Kautz, and Bernhard Rumpe. *MontiCore Language Workbench and Library Handbook: Edition 2021.* Aachener Informatik-Berichte, Software Engineering, Band 48. Shaker Verlag, May 2021.

[HLN+15a] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Composition of Heterogeneous Modeling Languages. In *Model-Driven Engineering and Software Development*, Communications in Computer and Information Science 580, pages 45–66. Springer, 2015.

[HLN+15] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, pages 19–31. SciTePress, 2015.

[HMR+19] Katrin Hölldobler, Judith Michael, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Innovations in Model-based Software and Systems Engineering. *Journal of Object Technology (JOT)*, 18(1):1–60, July 2019.

[HNRW16] Robert Heim, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Andreas Wortmann. Compositional Language Engineering using Generated, Extensible, Static Type Safe Visitors. In *Conference on Modelling Foundations and Applications (ECMFA)*, LNCS 9764, pages 67–82. Springer, July 2016.

[Hoe18] Katrin Hölldobler. *MontiTrans: Agile, modellgetriebene Entwicklung von und mit domänenspezifischen, kompositionalen Transformationssprachen.* Aachener Informatik-Berichte, Software Engineering, Band 36. Shaker Verlag, December 2018.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer Journal*, 37(10):64–72, October 2004.

[HR17] Katrin Hölldobler and Bernhard Rumpe. *MontiCore 5 Language Workbench Edition 2017.* Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017.

[HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling Dynamic Component Interfaces. In *Technology of Object-Oriented Languages and Systems (TOOLS 26)*, pages 58–70. IEEE, 1998.

[HRR10] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. Towards Architectural Programming of Embedded Systems. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VI*, Informatik-Bericht 2010-01, pages 13 – 22. fortiss GmbH, Germany, 2010.

[HRR+11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Software Product Lines Conference (SPLC'11)*, pages 150–159. IEEE, 2011.

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012.

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VII*, pages 1 – 10. fortiss GmbH, 2011.

[HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208. Springer, 2012.

[HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering Conference (SE'12)*, LNI 198, Seiten 181-192, 2012.

[HRW15] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 136–145. ACM/IEEE, 2015.

[HRW18] Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Software Language Engineering in the Large: Towards Composing and Deriving Languages. *Journal Computer Languages, Systems & Structures*, 54:386–405, 2018.

[JPR+22] Nico Jansen, Jerome Pfeiffer, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. The Language of SysML v2 under the Magnifying Glass. *Journal of Object Technology (JOT)*, 21:1–15, July 2022.

[JWCR18] Rodi Jolak, Andreas Wortmann, Michel Chaudron, and Bernhard Rumpe. Does Distance Still Matter? Revisiting Collaborative Distributed Software Design. *IEEE Software Journal*, 35(6):40–47, 2018.

[KER99] Stuart Kent, Andy Evans, and Bernhard Rumpe. UML Semantics FAQ. In A. Moreira and S. Demeyer, editors, *Object-Oriented Technology, ECOOP'99 Workshop Reader*, LNCS 1743, Berlin, 1999. Springer Verlag.

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM'09)*, Techreport B-108, pages 7–13. Helsinki School of Economics, October 2009.

[KKR19] Nils Kaminski, Evgeny Kusmenko, and Bernhard Rumpe. Modeling Dynamic Architectures of Self-Adaptive Cooperative Systems. *Journal of Object Technology (JOT)*, 18(2):1–20, July 2019.

[KKR+22] Jörg Christian Kirchhof, Anno Kleiss, Bernhard Rumpe, David Schmalzing, Philipp Schneider, and Andreas Wortmann. Model-driven Self-adaptive Deployment of Internet of Things Applications with Automated Modification Proposals. *Journal ACM Transactions on Internet of Things*, 3:1–30, November 2022.

[KKRZ19] Jörg Christian Kirchhof, Evgeny Kusmenko, Bernhard Rumpe, and Hengwen Zhang. Simulation as a Service for Cooperative Vehicles. In Loli Burgueño, Alexander Pretschner, Sebastian Voss, Michel Chaudron, Jörg Kienzle, Markus Völter, Sébastien Gérard, Mansooreh Zahedi, Erwan Bousse, Arend Rensink, Fiona Polack, Gregor Engels, and Gerti Kappel, editors, *Proceedings of MODELS 2019. Workshop MASE*, pages 28–37. IEEE, September 2019.

[KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Modelling of the Physical World Workshop (MOTPW'12)*, pages 2:1–2:6. ACM, October 2012.

[KMA+16] Jörg Kienzle, Gunter Mussbacher, Omar Alam, Matthias Schöttle, Nicolas Belloir, Philippe Collet, Benoit Combemale, Julien Deantoni, Jacques Klein, and Bernhard Rumpe. VCU: The Three Dimensions of Reuse. In *Conference on Software Reuse (ICSR'16)*, LNCS 9679, pages 122–137. Springer, June 2016.

[KMP+21] Hendrik Kausch, Judith Michael, Mathias Pfeiffer, Deni Raco, Bernhard Rumpe, and Andreas Schweiger. Model-Based Development and Logical AI for Secure and Safe Avionics Systems: A Verification Framework for SysML Behavior Specifications. In *Aerospace Europe Conference 2021 (AEC 2021)*. Council of European Aerospace Societies (CEAS), November 2021.

[KMR+20] Jörg Christian Kirchhof, Judith Michael, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Model-driven Digital Twin Construction: Synthesizing the Integration of Cyber-Physical Systems with Their Information Systems. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 90–101. ACM, October 2020.

[KMR21] Jörg Christian Kirchhof, Lukas Malcher, and Bernhard Rumpe. Understanding and Improving Model-Driven IoT Systems through Accompanying Digital Twins. In Eli Tilevich and Coen De Roover, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 21)*, pages 197–209. ACM, October 2021.

[KMS+18] Stefan Kriebel, Matthias Markthaler, Karin Samira Salman, Timo Greifenberg, Steffen Hillemacher, Bernhard Rumpe, Christoph Schulze, Andreas Wortmann, Philipp Orth, and Johannes Richenhagen. Improving Model-based Testing in Automotive Software Engineering. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE'18)*, pages 172–180. ACM, June 2018.

[KNP+19] Evgeny Kusmenko, Sebastian Nickels, Svetlana Pavlitskaya, Bernhard Rumpe, and Thomas Timmermanns. Modeling and Training of Neural Processing Systems. In Marouane Kessentini, Tao Yue, Alexander Pretschner, Sebastian Voss, and Loli Burgueño, editors, *Conference on Model Driven Engineering Languages and Systems (MODELS'19)*, pages 283–293. IEEE, September 2019.

[KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*, pages 284–297. IOS-Press, 1997.

[KPR12] Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In H. Lichter and B. Rumpe, Editoren, *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering, Band 14. Shaker Verlag, Aachen, Deutschland, 2012.

[KPRS19] Evgeny Kusmenko, Svetlana Pavlitskaya, Bernhard Rumpe, and Sebastian Stüber. On the Engineering of AI-Powered Systems. In Lisa O'Conner, editor, *ASE19. Software Engineering Intelligence Workshop (SEI19)*, pages 126–133. IEEE, November 2019.

[KR18a] Oliver Kautz and Bernhard Rumpe. On Computing Instructions to Repair Failed Model Refinements. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)*, pages 289–299. ACM, October 2018.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, März 2010.

[KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Workshop on Formal Methods for Open Object-based Distributed Systems*, IFIP Advances in Information and Communication Technology, pages 323–338. Chapmann & Hall, 1996.

[KRR14] Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe. *Trusted Cloud Computing*. Springer, Schweiz, December 2014.

[KRR+16] Philipp Kehrbusch, Johannes Richenhagen, Bernhard Rumpe, Axel Schloßer, and Christoph Schulze. Interface-based Similarity Analysis of Software Components for the Automotive Industry. In *International Systems and Software Product Line Conference (SPLC '16)*, pages 99–108. ACM, September 2016.

[KRRS19] Stefan Kriebel, Deni Raco, Bernhard Rumpe, and Sebastian Stüber. Model-Based Engineering for Avionics: Will Specification and Formal Verification e.g.

Based on Broy's Streams Become Feasible? In Stephan Krusche, Kurt Schneider, Marco Kuhrmann, Robert Heinrich, Reiner Jung, Marco Konersmann, Eric Schmieders, Steffen Helke, Ina Schaefer, Andreas Vogelsang, Björn Annighöfer, Andreas Schweiger, Marina Reich, and André van Hoorn, editors, *Proceedings of the Workshops of the Software Engineering Conference. Workshop on Avionics Systems and Software Engineering (AvioSE'19)*, CEUR Workshop Proceedings 2308, pages 87–94. CEUR Workshop Proceedings, February 2019.

[KRRW17] Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Modeling Architectures of Cyber-Physical Systems. In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 34–50. Springer, July 2017.

[KRS12] Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, Seiten 113-116. VDI Verlag, 2012.

[KRS+18a] Evgeny Kusmenko, Bernhard Rumpe, Sascha Schneiders, and Michael von Wenckstern. Highly-Optimizing and Multi-Target Compiler for Embedded System Models: C++ Compiler Toolchain for the Component and Connector Language EmbeddedMontiArc. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)*, pages 447 – 457. ACM, October 2018.

[KRS+22] Jörg Christian Kirchhof, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. MontiThings: Model-driven Development and Deployment of Reliable IoT Applications. *Journal of Systems and Software (JSS)*, 183:1–21, January 2022.

[KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Domain-Specific Modeling Workshop (DSM'06)*, Technical Report TR-37, pages 150–158. Jyväskylä University, Finland, 2006.

[KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Domain-Specific Modeling Workshop (DSM'07)*, Technical Reports TR-38. Jyväskylä University, Finland, 2007.

[KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'07)*, LNCS 4735, pages 286–300. Springer, 2007.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe'08)*, LNBIP 11, pages 297–315. Springer, 2008.

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

[KRW20] Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Automated semantics-preserving parallel decomposition of finite component and connector architectures. *Automated Software Engineering Journal*, 27:119–151, April 2020.

[Kus21] Evgeny Kusmenko. *Model-Driven Development Methodology and Domain-Specific Languages for the Design of Artificial Intelligence in Cyber-Physical Systems*. Aachener Informatik-Berichte, Software Engineering, Band 49. Shaker Verlag, November 2021.

[LMK+11] Philipp Leusmann, Christian Möllering, Lars Klack, Kai Kasugai, Bernhard Rumpe, and Martina Ziefle. Your Floor Knows Where You Are: Sensing and Acquisition of Movement Data. In Arkady Zaslavsky, Panos K. Chrysanthis, Dik Lun Lee, Dipanjan Chakraborty, Vana Kalogeraki, Mohamed F. Mokbel, and Chi-Yin Chow, editors, *12th IEEE International Conference on Mobile Data Management (Volume 2)*, pages 61–66. IEEE, June 2011.

[Loo17] Markus Look. *Modellgetriebene, agile Entwicklung und Evolution mehrbenutzerfähiger Enterprise Applikationen mit MontiEE*. Aachener Informatik-Berichte, Software Engineering, Band 27. Shaker Verlag, March 2017.

[LRSS10] Tihamer Levendovszky, Bernhard Rumpe, Bernhard Schätz, and Jonathan Sprinkle. Model Evolution and Management. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 241–270. Springer, 2010.

[MKB+19] Felix Mannhardt, Agnes Koschmider, Nathalie Baracaldo, Matthias Weidlich, and Judith Michael. Privacy-Preserving Process Mining: Differential Privacy for Event Logs. *Business & Information Systems Engineering*, 61(5):1–20, October 2019.

[MKM+19] Judith Michael, Agnes Koschmider, Felix Mannhardt, Nathalie Baracaldo, and Bernhard Rumpe. User-Centered and Privacy-Driven Process Mining System Design for IoT. In Cinzia Cappiello and Marcela Ruiz, editors, *Proceedings of CAiSE Forum 2019: Information Systems Engineering in Responsible Information Systems*, pages 194–206. Springer, June 2019.

[MM13] Judith Michael and Heinrich C. Mayr. Conceptual modeling for ambient assistance. In *Conceptual Modeling - ER 2013*, LNCS 8217, pages 403–413. Springer, 2013.

[MM15] Judith Michael and Heinrich C. Mayr. Creating a domain specific modelling method for ambient assistance. In *International Conference on Advances in ICT for Emerging Regions (ICTer2015)*, pages 119–124. IEEE, 2015.

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer Journal*, 43(5):42–48, May 2010.

[MMR+17] Heinrich C. Mayr, Judith Michael, Suneth Ranasinghe, Vladimir A. Shekhovtsov, and Claudia Steinberger. *Model Centered Architecture*, pages 85–104. Springer International Publishing, 2017.

[MNRV19] Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Towards Privacy-Preserving IoT Systems Using Model Driven Engineering. In Nicolas Ferry, Antonio Cicchetti, Federico Ciccozzi, Arnor Solberg, Manuel Wimmer, and Andreas Wortmann, editors, *Proceedings of MODELS 2019. Workshop MDE4IoT*, pages 595–614. CEUR Workshop Proceedings, September 2019.

[MPRW22] Judith Michael, Jérôme Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. Integration Challenges for Digital Twin Systems-of-Systems. In *10th IEEE/ACM International Workshop on Software Engineering for Systems-of-Systems and Software Ecosystems*, pages 9–12. ACM, May 2022.

[MRR10] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. A Manifesto for Semantic Model Differencing. In *Proceedings Int. Workshop on Models and Evolution (ME'10)*, LNCS 6627, pages 194–203. Springer, 2010.

[MRR11d] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Conference on Foundations of Software Engineering (ESEC/FSE '11)*, pages 179–189. ACM, 2011.

[MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011.

[MRR11e] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 592–607. Springer, 2011.

[MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CDDiff: Semantic Differencing for Class Diagrams. In Mira Mezini, editor, *ECOOP 2011 - Object-Oriented Programming*, pages 230–254. Springer Berlin Heidelberg, 2011.

[MRR11c] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *Object-Oriented Programming Conference (ECOOP'11)*, LNCS 6813, pages 281–305. Springer, 2011.

[MRR11f] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 153–167. Springer, 2011.

[MRR11g] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Summarizing Semantic Model Differences. In Bernhard Schätz, Dirk Deridder, Alfonso Pierantonio, Jonathan Sprinkle, and Dalila Tamzalit, editors, *ME 2011 - Models and Evolution*, October 2011.

[MRR13] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views. In Meyer, B. and Baresi, L. and Mezini, M., editor, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, pages 444–454. ACM New York, 2013.

[MRR14a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views (extended abstract). In Wilhelm Hasselbring and Nils Christian Ehmke, editors, *Software Engineering 2014*, LNI 227, pages 63–64. Gesellschaft für Informatik, Köllen Druck+Verlag GmbH, 2014.

[MRR14b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *International Conference on Software Engineering (ICSE'14)*, pages 95–105. ACM, 2014.

[MRRW16] Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe, and Michael von Wenckstern. Consistent Extra-Functional Properties Tagging for Component and Connector Models. In *Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp'16)*, CEUR Workshop Proceedings 1723, pages 19–24, October 2016.

[MRV20] Judith Michael, Bernhard Rumpe, and Simon Varga. Human behavior, goals and model-driven software engineering for assistive systems. In Agnes Koschmider, Judith Michael, and Bernhard Thalheim, editors, *Enterprise Modeling and Information Systems Architectures (EMSIA 2020)*, pages 11–18. CEUR Workshop Proceedings, June 2020.

[MRZ21] Judith Michael, Bernhard Rumpe, and Lukas Tim Zimmermann. Goal Modeling and MDSE for Behavior Assistance. In *Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 370–379. ACM/IEEE, October 2021.

[MS17] Judith Michael and Claudia Steinberger. Context modeling for active assistance. In Cristina Cabanillas, Sergio España, and Siamak Farshidi, editors, *Proc. of the ER Forum 2017 and the ER 2017 Demo Track co-located with the 36th Int. Conference on Conceptual Modelling (ER 2017)*, pages 221–234, 2017.

[Naz17] Pedram Mir Seyed Nazari. *MontiCore: Efficient Development of Composed Modeling Language Essentials.* Aachener Informatik-Berichte, Software Engineering, Band 29. Shaker Verlag, June 2017.

[NRR15a] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. Mixed Generative and Handcoded Development of Adaptable Data-centric Business Applications. In *Domain-Specific Modeling Workshop (DSM'15)*, pages 43–44. ACM, 2015.

[NRR16] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. An Extended Symbol Table Infrastructure to Manage the Composition of Output-Specific Generator Information. In *Modellierung 2016 Conference*, LNI 254, pages 133–140. Bonner Köllen Verlag, March 2016.

[PR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, CEUR Workshop Proceedings 1118, pages 15–24, 2013.

[PBI+16] Dimitri Plotnikov, Inga Blundell, Tammo Ippen, Jochen Martin Eppler, Abigail Morrison, and Bernhard Rumpe. NESTML: a modeling language for spiking neurons. In *Modellierung 2016 Conference*, LNI 254, pages 93–108. Bonner Köllen Verlag, March 2016.

[PFR02] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product Line Annotations with UML-F. In *Software Product Lines Conference (SPLC'02)*, LNCS 2379, pages 188–197. Springer, 2002.

[Pin14] Claas Pinkernell. *Energie Navigator: Software-gestützte Optimierung der Energieeffizienz von Gebäuden und technischen Anlagen.* Aachener Informatik-Berichte, Software Engineering, Band 17. Shaker Verlag, 2014.

[Plo18] Dimitri Plotnikov. *NESTML - die domänenspezifische Sprache für den NEST-Simulator neuronaler Netzwerke im Human Brain Project.* Aachener Informatik-Berichte, Software Engineering, Band 33. Shaker Verlag, February 2018.

[PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *Proceedings of the Industrial Benefit of Formal Methods (FME'94)*, LNCS 873, pages 154–174. Springer, 1994.

[PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In *Congress on Formal Methods in the Development of Computing System (FM'99)*, LNCS 1708, pages 96–115. Springer, 1999.

[PR01] Jan Philipps and Bernhard Rumpe. Roots of Refactoring. In Kilov, H. and Baclavski, K., editor, *Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15.* Northeastern University, 2001.

[PR03] Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In Kilov, H. and Baclavski, K., editor, *Practical Foundations of Business and System Specifications*, pages 281–297. Kluwer Academic Publishers, 2003.

[Rei16] Dirk Reiß. *Modellgetriebene generative Entwicklung von Web-Informationssystemen.* Aachener Informatik-Berichte, Software Engineering, Band 22. Shaker Verlag, May 2016.

[Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems.* Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, Aachen, Germany, December 2014.

[RK96] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In B. Harvey and H. Kilov, editors, *Object-Oriented Behavioral Specifications*, pages 265–286. Kluwer Academic Publishers, 1996.

[RKB95] Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technischer Bericht TUM-I9510, TU München, Deutschland, März 1995.

[Rot17] Alexander Roth. *Adaptable Code Generation of Consistent and Customizable Data Centric Applications with MontiDex.* Aachener Informatik-Berichte, Software Engineering, Band 31. Shaker Verlag, December 2017.

[RR11] Jan Oliver Ringert and Bernhard Rumpe. A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. *International Journal of Software and Informatics*, 2011.

[RRRW15b] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 6(1):33–57, 2015.

[RRS+16] Johannes Richenhagen, Bernhard Rumpe, Axel Schloßer, Christoph Schulze, Kevin Thissen, and Michael von Wenckstern. Test-driven Semantical Similarity Analysis for Software Product Line Extraction. In *International Systems and Software Product Line Conference (SPLC '16)*, pages 174–183. ACM, September 2016.

[RRSW17] Jan Oliver Ringert, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. Teaching Agile Model-Driven Engineering for Cyber-Physical Systems. In *International Conference on Software Engineering: Software Engineering and Education Track (ICSE'17)*, pages 127–136. IEEE, May 2017.

[RRW12] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Requirements Modeling Language for the Component Behavior of Cyber Physical Robotics Systems. In Seyff, N. and Koziolek, A., editor, *Modelling and Quality in Requirements Engineering: Essays Dedicated to Martin Glinz on the Occasion of His 60th Birthday*, pages 133–146. Monsenstein und Vannerdat, Münster, 2012.

[RRW13] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In *Software Engineering Workshopband (SE'13)*, LNI 215, pages 155–170, 2013.

[RRW13c] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013.

[RRW14a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Aachener Informatik-Berichte, Software Engineering, Band 20. Shaker Verlag, December 2014.

[RRW15] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Tailoring the MontiArcAutomaton Component & Connector ADL for Generative Development. In *MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*, pages 41–47. ACM, 2015.

[RSW+15] Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, Jan Oliver Ringert, and Peter Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *Software Product Line Conference (SPLC'15)*, pages 141–150. ACM, 2015.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996.

[Rum02] Bernhard Rumpe. Executable Modeling with UML - A Vision or a Nightmare? In T. Clark and J. Warmer, editors, *Issues & Trends of Information Technology Management in Contemporary Associations, Seattle*, pages 697–701. Idea Group Publishing, London, 2002.

[Rum03] Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In *Symposium on Formal Methods for Components and Objects (FMCO'02)*, LNCS 2852, pages 380–402. Springer, November 2003.

[Rum04c] Bernhard Rumpe. Agile Modeling with the UML. In *Workshop on Radical Innovations of Software and Systems Engineering in the Future (RISSEF'02)*, LNCS 2941, pages 297–309. Springer, October 2004.

[Rum11] Bernhard Rumpe. *Modellierung mit UML, 2te Auflage*. Springer Berlin, September 2011.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring, 2te Auflage*. Springer Berlin, Juni 2012.

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.

[Rum17] Bernhard Rumpe. *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International, May 2017.

[RW18] Bernhard Rumpe and Andreas Wortmann. Abstraction and Refinement in Hierarchically Decomposable and Underspecified CPS-Architectures. In Lohstroh, Marten and Derler, Patricia Sirjani, Marjan, editor, *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*, LNCS 10760, pages 383–406. Springer, 2018.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[SHH+20] Günther Schuh, Constantin Häfner, Christian Hopmann, Bernhard Rumpe, Matthias Brockmann, Andreas Wortmann, Judith Maibaum, Manuela Dalibor, Pascal Bibow, Patrick Sapel, and Moritz Kröger. Effizientere Produktion mit Digitalen Schatten. *ZWF Zeitschrift für wirtschaftlichen Fabrikbetrieb*, 115(special):105–107, April 2020.

[SM18a] Claudia Steinberger and Judith Michael. Towards Cognitive Assisted Living 3.0. In *International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops 2018)*, pages 687–692. IEEE, march 2018.

[SM20] Claudia Steinberger and Judith Michael. Using Semantic Markup to Boost Context Awareness for Assistive Systems. In *Smart Assisted Living: Toward An Open Smart-Home Infrastructure*, Computer Communications and Networks, pages 227–246. Springer International Publishing, 2020.

[SRVK10] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling: State of the Art and Research Challenges. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 57–76. Springer, 2010.

[TAB+21] Carolyn Talcott, Sofia Ananieva, Kyungmin Bae, Benoit Combemale, Robert Heinrich, Mark Hills, Narges Khakpour, Ralf Reussner, Bernhard Rumpe, Patrizia Scandurra, and Hans Vangheluwe. Composition of Languages, Models, and Analyses. In Heinrich, Robert and Duran, Francisco and Talcott, Carolyn and Zschaler, Steffen, editor, *Composing Model-Based Analysis Tools*, pages 45–70. Springer, July 2021.

[THR+13] Ulrike Thomas, Gerd Hirzinger, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *Conference on Robotics and Automation (ICRA'13)*, pages 461–466. IEEE, 2013.

[Voe11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen.* Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.

[WCB17] Andreas Wortmann, Benoit Combemale, and Olivier Barais. A Systematic Mapping Study on Modeling for Industry 4.0. In *Conference on Model Driven Engineering Languages and Systems (MODELS'17)*, pages 281–291. IEEE, September 2017.

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen.* Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012.

[Wor16] Andreas Wortmann. *An Extensible Component & Connector Architecture Description Infrastructure for Multi-Platform Modeling.* Aachener Informatik-Berichte, Software Engineering, Band 25. Shaker Verlag, November 2016.

[Wor21] Andreas Wortmann. *Model-Driven Architecture and Behavior of Cyber-Physical Systems.* Aachener Informatik-Berichte, Software Engineering, Band 50. Shaker Verlag, October 2021.

[ZPK+11] Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In *Proceedings of the SESAR Innovation Days.* EUROCONTROL, 2011.