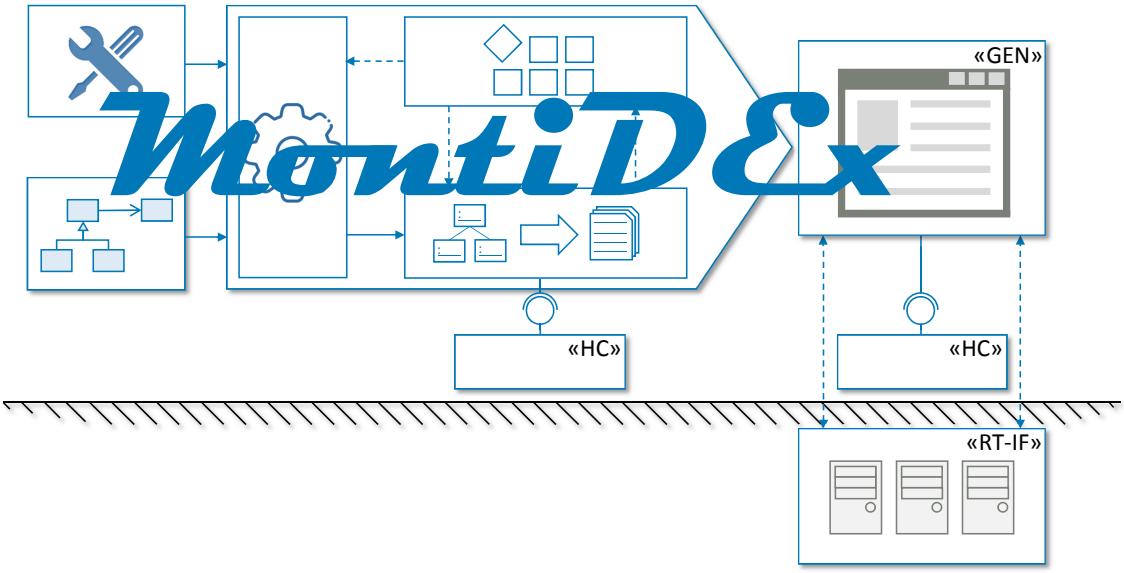
SE | RWTH AACHEN UNIVERSITY

Alexander Roth

Adaptable Code Generation of Consistent and Customizable Data-Centric Applications with MontiDEx



Aachener Informatik-Berichte,
Software Engineering

Band 31

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

[Rot18] A. Roth:
Adaptable Code Generation of Consistent and Customizable Data Centric Applications with MontiDex.
Shaker Verlag, ISBN 978-3-8440-5688-4. Aachener Informatik-Berichte, Software Engineering, Band 31. December 2017.
www.se-rwth.de/publications/

Adaptable Code Generation of Consistent and Customizable Data-Centric Applications with MontiDEx

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

M.Sc. RWTH
Alexander Roth
aus Ujar, Russland

Berichter: Universitätsprofessor Dr. rer. nat. Bernhard Rumpe
Universitätsprofessor Dr. rer. nat. Albert Zündorf

Tag der mündlichen Prüfung: 15. November 2017

Abstract

Information systems are software systems that address current demands for harvesting, storing, and manipulating structured information [Kaj12]. A part of information systems are client applications (subsequently called *data-centric applications*) with a graphical user interface (GUI) to execute predefined CRUD (Create, Read, Update, and Delete) operations and display the managed data. The development and prototyping of such software systems contain common repetitive and error-prone development tasks such as implementing a *data structure*, which is a source-code-representation of the managed information, realizing a GUI, and providing access to a persistence infrastructure.

Model-driven development (MDD) aims to reduce the development effort of data-centric applications, improve software quality, and reduce development costs [BCW12] by exploiting platform-independent models as primary development artifacts [SVC06]. Each model is an instance of a domain-specific language (DSL), which enables a high-level and abstract description of (parts of) a software system using domain terminology. MDD tools systematically transform a model into executable source code.

Nevertheless, effective MDD of data-centric prototypes has to address several challenges. In particular, fully executable prototypes have to be generated including the data structure and GUI, which is necessary to support information gathering from end users [MFM+13]. Moreover, adequate DSLs have to be provided to describe structured information and facilitate underspecification that enables prototyping in early stages of the development. These concerns are additionally influenced by orthogonal customization concerns of the generated source code to support agile software development (cf. [BPRFF15]).

Similar holds for MDD of data-centric development, where the same challenges have to be addressed. In addition, an MDD approach has to avoid additional overhead introduced by maintaining DSLs and MDD tools (cf. [MK09]) to improve developer acceptance (cf. [KBR11]). Furthermore, such an approach has to provide adaptation mechanisms for MDD tools to facilitate MDD tool reuse by enabling framework-like and standalone use.

To address the aforementioned challenges, this thesis contributes methods and concepts to a lightweight and agile MDD approach for data-centric application development and its data structure prototyping. The goal is to improve development efficiency by reducing the necessary manual implementation effort for repetitive implementation tasks. In particular, implementing a GUI, a data structure that ensures data consistency, a connection to a persistence infrastructure, and support for process automation, which is considered as the automated execution of CRUD operations. This is achieved by a language family for structural and behavioral modeling, code generators, and lightweight methods. Since such an approach is always accompanied by customization, adaptation, and extension concerns (cf. [SVC06]), mechanisms for manually-written customizations

of generated source code, as well as a modular and adaptable code generator design to facilitate code generator reuse are addressed (cf. [ZR11]). Furthermore, a method and technical realization for all proposed concepts is provided by the MontiCore Data Explorer (MontiDEx) code generator.

Employing the methods proposed in this thesis improves development and prototyping of data-centric applications by providing a unified set of languages and lightweight methods abstracting from implementation details and supporting customization and adaptation concerns of generated source code and MDD tools.

Kurzfassung

Informationssysteme sind Softwaresysteme, die das Verwalten von strukturierten Daten unterstützen und fördern (vgl. [Kaj12]). Ein Teil solcher Informationssysteme sind Client-Anwendungen (im weiteren Verlauf *datenzentrische Anwendungen* genannt). Diese besitzen eine graphische Benutzeroberfläche und erlauben das Ausführen von CRUD (Create, Read, Update und Delete) Operationen auf den verwalteten Daten, sowie das Anzeigen der Resultate. Jedoch beinhaltet die Entwicklung und das Prototyping solcher datenzentrischen Anwendungen viele gemeinsame und sich wiederholende Entwicklungsaufgaben, wie z.B. das Implementieren einer Datenstruktur, die die Quellcode-Repräsentation der verwalteten Information darstellt, das Realisieren einer graphischen Benutzeroberfläche und das Anbieten einer Schnittstelle zu einer Persistenzinfrastruktur für die Datenspeicherung.

Ein Ansatz zur Reduzierung dieser sich wiederholenden Entwicklungsaufgaben zur Steigerung der Softwarequalität und zur Senkung der Entwicklungskosten bietet die Modellgetriebene Softwareentwicklung (MDD) (vgl. [BCW12]). Dies geschieht dadurch, dass plattformunabhängige Modelle als primäres Entwicklungsartefakt genutzt werden (vgl. [SVC06]). Jedes Modell ist dabei eine Instanz einer domänenspezifischen Sprache (DSL), welche ein hohes Abstraktionsniveau bietet um Teile oder ganze Softwaresysteme zu beschreiben und gleichzeitig Domänenterminologie zu verwenden. Werkzeuge transformieren systematisch ein Modell in ausführbaren Quellcode.

Dennoch muss ein effektiver Ansatz zur modellgetriebenen Softwareentwicklung von datenzentrischen Prototypen mehrere Herausforderungen bewältigen. Eine Herausforderung ist die Generierung von vollständig ausführbaren Prototypen. Dies ist Notwendig um die Informationsgewinnung von Endbenutzern zu unterstützen (vgl. [MFM+13]). Diese ausführbaren Prototypen müssen die Datenstruktur beinhalten und eine graphische Benutzeroberfläche anbieten. Bei der modellgetriebenen Softwareentwicklung setzt dies jedoch voraus, dass geeignete DSLs zur Verfügung gestellt werden, die es ermöglichen strukturierte Informationen abstrakt zu beschreiben und auch während der frühen Phase des Prototypings, welche auch Unterspezifikation beinhaltet, genutzt werden können. Eine weitere Herausforderung ist die Bereitstellung von Möglichkeiten zur Anpassung der generierten datenzentrischen Anwendung, die vor allem in einer agilen Entwicklungsumgebung erforderlich sind (vgl. [BPRFF15]).

Gleiche Herausforderungen existieren auch in der modellgetriebene Softwareentwicklung von datenzentrischen Anwendungen. Jedoch sollte zusätzlich der gewählte MDD Ansatz Overhead vermeiden, der durch die Wartung und die Entwicklung von DSLs und geeigneten Werkzeugen eingeführt wird (vgl. [MK09]). Die Vermeidung dieses Overheads steigert die Akzeptanz der Entwickler (vgl. [KBR11]). Darüber hinaus muss ein derartiger Ansatz Anpassungsmechanismen für die genutzten Werkzeuge bereitstellen, um deren Wiederverwendung zu fördern.

Um diese Herausforderungen zu adressieren, werden in dieser Arbeit leichtgewichtige Methoden und Konzepte für ein modellgetriebenes Datenstruktur-Prototyping und die modellgetriebene Softwareentwicklung von datenzentrischen Anwendungen vorgestellt. Ziel ist es, die Effizienz in der Entwicklung zu verbessern, indem der notwendige manuelle Implementierungsaufwand für sich wiederholende Implementierungsaufgaben reduziert wird. Dabei steht insbesondere der Implementierungsaufwand bei der Entwicklung einer graphischen Benutzeroberfläche, einer Datenstruktur und einer Anbindung an eine Persistenzinfrastruktur im Fokus dieser Arbeit. Dies wird durch eine geeignete Sprachfamilie für die Struktur und Prozessbeschreibung von datenzentrischen Anwendungen, geeignete Code-Generatoren und leichtgewichtige Entwicklungsmethoden erreicht. Um Anpassungen im generierten Produkt als auch im Generator zu ermöglichen, werden Methoden zur handgeschriebenen Erweiterung und Konzepte zur Realisierung eines modularen Code-Generators vorgestellt, welche die Wiederverwendung fördern (vgl. [ZR11]). Die entwickelten Konzepte wurden im MontiCore Data Explorer (MontiDEx) Code Generator, sowie dem MontiDEx Produkt realisiert.

Danksagung

In der Entstehungszeit meiner Promotion wurde ich von vielen lieben Menschen unterstützt und begleitet, die durch hilfreiche Diskussionen, tatkräftige Unterstützung und notwendige Ablenkung zum Erfolg dieser Promotion beigetragen haben. Deshalb möchte ich an dieser Stelle diesen Menschen meinen tiefsten Dank aussprechen.

Mein besonderer Dank gilt meinem Doktorvater Prof. Dr. Bernhard Rumpe für die Betreuung dieser Promotion und die spannende und abwechslungsreiche Zeit am Lehrstuhl. Die vielen konstruktiven und lebhaften Diskussionen sowie die vielen Ratschläge haben den Erfolg dieser Promotion entscheidend bewirkt. Auch danke ich dir sehr, Bernhard, dass du mir die Möglichkeit gegeben hast frühzeitig die Leitung verschiedener Forschungs- und Industrieprojekte zu übernehmen und mich so in meiner persönlichen Entwicklung unterstützt hast. Weiterhin bedanke ich mich bei dir für dein Vertrauen mir die Gruppenleitung der Digitalisierungsgruppe zu übergeben.

Weiterer Dank gilt Prof. Dr. Albert Zündorf für das Interesse an meiner Arbeit, das sehr gute Feedback und die Übernahme meines Zweitgutachtens. Ebenfalls möchte ich mich bei Prof. Dr. Ulrik Schröder für die Leitung meines Promotionskomitees und die Abnahme der Prüfung in der praktischen Informatik bedanken. Herrn apl. Prof. Dr. Thomas Noll danke ich für die Abnahme der Prüfung in der theoretischen Informatik.

Herzlich bedanke ich mich auch bei allen Kollegen, die mich während meiner Zeit am Lehrstuhl (oftmals auch außerhalb) begleitet haben. Insbesondere möchte ich Dr. Martin Schindler für sein Mentoring am Anfang meiner Promotionszeit danken. Ebenfalls danke ich Antonio Navarro Pérez für die viele Ideen und den großartigen Einfluss auf meine Promotion. Mein besonderer Dank gilt meinem ehemaligen Gruppenleiter Dr. Pedram Mir Seyed Nazari. Wir haben zusammen den Weihnachtsmann überlebt und konnten immer über jegliche Themen lebhaft und lange diskutieren. Danke für die gemeinsame Zeit, die Zusammenarbeit in der Forschung und Lehre und die Motivation während meiner Promotion! Dr. Klaus Müller danke ich für die sehr gute Zusammenarbeit an verschiedenen Publikationen und die vielen gemeinsamen Forschungsworkshops. Auch wenn du als externer selten anwesend warst, war die Zusammenarbeit mit dir sehr fruchtbar und erfolgreich. Herrn "von Wenckstern" danke ich für seine offene, lustige und immer unterhaltsame Art mit der er das Leben am Lehrstuhl aufgeheitert hat. Gleichzeitig danke ihm auch für die sehr intensiven und hilfreichen Diskussionen. Evgeny Kusmenko möchte ich für seine Motivation zur sportlichen Betätigung danken, aber auch für die gute Zusammenarbeit. Ich bedanke mich auch sehr bei Kai Adam, der mich stark in der letzten Phase meiner Promotion und in unterschiedlichen Forschungs- und Industrieprojekten tatkräftig unterstütz hat.

Darüber hinaus danke ich Sylvia Gunder und Gabriele Heuschen, die mich bei allen organisatorischen Aufgaben unterstützt haben und mein Anlaufpunkt bei Unklarheiten waren. Ebenfalls bedanke ich mich bei Galina Volkova und Marita Breuer, die im Rah-

men des MontiCore Projektes ihr Wissen gerne geteilt haben und mich auch während vieler Demonstrationen und Implementierungsarbeiten unterstütz haben. Außerdem danke ich allen Wissenschaftlichen Mitarbeitern des Lehrstuhls: Vincent Bertram, Arvid Butting, Robert Eikermann, Timo Greifenberg, Dr. Arne Haber, Robert Heim, Steffen Hillemacher, Lars Hermerschmidt, Andreas Horst, Katrin Hölldobler, Oliver Kautz, Carsten Kolassa, Thomas Kurpick, Achim Lindt, Matthias Markthaler, Dimitri Plotnikov, Deni Raco, Dr. Jan Oliver Ringert, Steffi Schrader, Christoph Schulze und Dr. Andreas Wortmann. Für die tatkräftige Unterstützung bedanke ich mich auch bei allen Auszubildenden: Lennart Bucher, Manuel Pütz, Jerome Pfeifer, Ben Mainz, Brian Sinkovec und Max Voss. Vielen Dank auch an all diejenigen, die meine Arbeit Korrektur gelesen haben: Kai Adam, Arvid Butting, Robert Heim, Oliver Kautz, Matthias Markthaler, Dr. Klaus Müller, Dr. Pedram Mir Seyed Nazari, Max Voß und Michael von Wenckstern.

Natürlich gilt mein Dank auch allen Studenten und Studentinnen, die an der Promotion in Form von studentischen Abschlussarbeiten tatkräftig mitgewirkt haben: Aydin Alatas, Denis Domm, Ralph Geerkens, Thomas Maiwald, Patrick Jan Schlesiona, Dominik Studer, Philip Uhl und Enis Zejnilovic. Insbesondere danke ich Junior Lekane Nimpa für die lange Zusammenarbeit an MontiDEx und die sehr guten Implementierungs- und Gestaltungsmitarbeiten.

Abschließend gilt mein unendlicher Dank meinen Eltern und meinen Freunden, die mich auf meinem bisherigen Lebensweg begleitet und unterstützt haben. Liebe Eltern, Peter und Katharina Roth, ich danke euch von ganzem Herzen. Ihr habt mir die Motivation und das nötige Durchhaltevermögen gegeben und als Vorbilder gezeigt, wie man seine Ziele erreichen kann. Diese Promotion wäre ohne eure Aufopferung, Liebe und Unterstützung nicht möglich gewesen. Ebenfalls bedanke ich mich auch bei meiner Schwester, Olga Knabe, die mir auch während schwieriger Zeiten immer mit Rat und Tat zur Seite stand, mich immer unterstütz und mit kleinen Geschenken motiviert hat. Auch möchte ich Dr. Bastian Knabe und Melissa Anastasia Knabe danken für die Aufheiterung und das Interesse an meiner Promotion. Ich danke auch Michelle Egge für die grenzenlose Unterstützung und unglaubliche Geduld. Für die notwendige Ablenkung und Erholung möchte ich euch, Hauke und Lisa Schaper, ganz herzlich danken. Schließlich danke ich auch Florian Kerber, Michael Lutz und Dr. Andreas Ganser für die Begleitung auf meinem bisherigen Lebensweg und die schöne Zeit mit euch.

Aachen, November 2017
Alexander Roth

Contents

Chapter 1

Introduction

Due to current trends such as Industry 4.0 [Gil16], Internet of Things (IoT) [AIM10], and Big Data [CML14], harvesting, storing, and manipulating structured information has become an essential concern in software development. The term "structured" emphasizes the property of information to be decomposed into interconnected conceptual units. The manipulation by predefined CRUD (Create, Read, Update, and Delete operations) operations and persistent storage of structured information is the main concern of an *information system (InfoSys)* [Kaj12]. A part of an InfoSys are lightweight client applications with a graphical user interface (GUI) to execute the predefined CRUD operations and display the stored information; provide access to a *persistence infrastructure*, which stores managed data; and support process automation, which, in this thesis, is considered as the automated execution of CRUD operations on the managed data. In this thesis, such lightweight client applications are considered as *data-centric applications*.

Among others, an important task of data-centric application development - that is part of most software projects [HL01] - is prototyping, which is the development of a first version (i.e., *prototype*) of the product for demonstration purposes [Som10]. For example, such a prototype can be used to consolidate the requirements with the end user (cf. Section 3.1.3) regarding the information to be managed. The development of a data-centric application prototype involves, among others, the implementation of a GUI and a source-code-representation of the structured information (subsequently called *data structure*), which is the mapping of the structured information to classes, attributes, and methods in a *general purpose programming language (GPL)*. A working prototype is necessary, because "information gathering is most effective if it is based on something that works" [MFM+13]. Manually implementing a data-centric application prototype is a repetitive, time-consuming, and error-prone task, which has to be repeated multiple times, because the prototype is iteratively improved and evaluated with end users of the final product [Som10]. An additional drawback of manually implementing prototypes is that the structured information being prototyped is not used as a primary development artifact such as, e.g., source code.

A data-centric application *product* (short: data-centric application) is developed, based on the requirements identified during prototyping. The development of a data-centric application involves, among others, implementing a GUI, a data structure, and

provide access to a persistence infrastructure, which facilitates management of multiple users, and implementation of support for process automation. In addition, the data structure has to ensure *data consistency*, i.e., the constraints in the structured information are not violated. This is necessary to reduce the gap between the structured information's description and its implementation. These development tasks are a superset of the development tasks needed for data-centric application prototype development. Hence, manual development of data-centric applications yields at least the same drawbacks as data-centric application prototyping. To reduce these drawbacks, a data-centric application prototype can be used to support implementation tasks [BPRFF15]. In particular, reuse of artifacts, which are developed during prototyping, saves time and effort [MFM+13] and may even yield a productivity improvement in the order-of-magnitude [vL00].

Model-driven development (MDD) aims to reduce manual development effort, improve software quality, and reduce development costs [BCW12] by employing abstract and platform-independent models as means to describe (parts of) the software systems. For example, the structured information managed by a data-centric application can be described by UML class diagram (UML CD) models [www15b] and processes can be described by UML activity diagram (UML AD) models [www15b]. Each abstract model, which is an instance of a *domain-specific language (DSL)*[1], is systematically transformed into platform-dependent source code (subsequently called *generated source code*), i.e., the input language's concepts are mapped to target language source code using a *mapping*, by corresponding MDD tools (*MDSD-Tools* in [SVC06]). The generated source code represents either a part of a software system, e.g., the data structure of a data-centric application, or the whole software system, e.g., data-centric application.

Existing approaches for prototyping of data-centric applications (subsequently called *model-driven prototyping (MDP)*) mainly address prototyping of GUIs [MP03, MP04, MFM+13] or prototyping of use cases [LLHL05, LL08] without generating fully functional prototypes for data structure prototyping. However, not fully functional prototypes hamper requirement elucidation of the data structure. Other approaches are restricted in means to describe data structures to 1-to-many and 1-to-1 relations [MSHL06], which is a rather hard restriction that omits the most commonly used relations (i.e., all combinations of `[1]`, `[0..1]`, `[1..*]`, and `[*]` [BFL13]). Finally, other approaches omit underspecification in data structure description [FBLS12, Let14a, Let14b, GAL15], which hampers prototyping in early stages of development, where requirements are vague but a prototype has to be developed.

Similar issues hold for MDD of data-centric applications: large scale MDD approaches (i.e., development of information systems) [BHKN96, KVR02, Dog08, Sub15] introduce

[1] A DSL is a "language that is specifically dedicated to a domain of interest" [CCF+15], where a language is a means for communication between stakeholders and machines. It restricts the amount of sentences that can be communicated [CCF+15]. In the remainder of this thesis, we use DSL and *domain-specific modeling language (DSML)* (short *modeling language (ML)*) interchangeably, because they cannot be separated [Loo17]. A more detailed explanation is given in Section 2.1.1.

an overhead of maintaining DSLs and MDD tools (cf. [MK09]) to describe and develop the overall software system, which has led to reduced acceptance (cf. [KBR11]); address teaching purposes [PBCN15] or MDD of GUIs only [Van05, Aki13, JC15] generating a code frame that has to be manually extended; are limited in configuration and adaptation mechanisms of MDD tools to building blocks [KR08] and modular templates [Sol10] and, hence, do not facilitate black-box extension and adaptation of MDD tool to improve artifact reuse from MDP; and only facilitate framework-like use with a minimal GUI and minimal configuration and adaptation capabilities [Lan16].

Hence, the goal of this thesis is to contribute methods and concepts to a lightweight and agile MDP and MDD approach for data-centric applications that differ from existing lightweight approaches in terms of:

- Applicability to MDP and MDD of data-centric applications.
- Support for underspecification in data structure descriptions.
- Mechanisms for configuration, customization, and adaptation of the generated source code and MDD tools.

In particular, this thesis presents:

- A *language family*[2] to model structured information (i.e, *analysis model* in MDP and *domain model* in MDD of data-centric applications[3]); and to specify processes.
- A mapping from analysis models to Java source code (i.e., the data structure). It extends existing mappings [Rum12, BFL13] to ensure data consistency.
- A *data-centric infrastructure* extending the data structure to a data-centric application or prototype and enables modular and framework-like use and extension.
- Explicit mechanisms to customize the generated source code as well as the data-centric infrastructure.
- Adaptation and extension mechanisms for MDD tools.
- An extended data-centric infrastructure supporting process automation.

[2] A language family is a collection of languages, each of which models a different system aspect, that can be interpreted together [Wor16].

[3] In this thesis, analysis and domain models are lightweight UML CD models, which describe the structured information of a data-centric application (cf. Section 4.1). In this thesis, they are considered to use the same UML CD language concepts but are distinguished by name because analysis models are used for MDP, whereas domain models are used for MDD of data-centric applications. Hence, we use the terms interchangeably.

This first chapter introduces the context of this thesis in Section 1.1. Afterwards, the thesis' objectives and contributions are described in Section 1.2 and the thesis' structure is explained in Section 1.3. Finally, an overview of related publications created in the course of this thesis are presented in Section 1.4.

1.1 Context of the Thesis

The foundation for the developed concepts and tools is formed by previous research of the chair of Software Engineering at the RWTH Aachen University. In more detail, the DSL for describing structured information is founded on the UML/P class diagram (UML/P CD) ML [Sch12, Rum16] but with language concept restrictions such as omitting methods, constructors, and visibility. A restriction of language concepts is necessary to support the language's successful use (cf. [KKP+09]) as means to describe analysis models. The language for process description is an extended variant of an already proposed DSL [Rei15] with additional concepts to simplify control/object flows, explicit function-like notion for input and output pins, and an auto-connect approach to automatically connect input and output pins based on their type and name, which is founded on the auto-connect capabilities of MontiArc [Hab15]. The modular design of the generated data-centric infrastructure is based on approaches for modular infrastructures [Her13, Loo17]. Finally, the persistence infrastructure is founded on concepts for enterprise application development [Loo17].

The technical realization of this thesis is based on the MontiCore (MC) language workbench and code generation framework (cf. Section 2.2). In particular, it has been used for the development of:

- The *UML class diagram for analysis (CD4A)* ML, which is a DSL for analysis models developed in this thesis.
- The *UML class diagram for code (CD4Code)* ML, which is a DSL used in the code generation approach as an intermediate representation developed in this thesis.
- The *UML activity diagram with embedded JavaDSL (ADJava)* ML, which is a DSL for process description developed in this thesis.
- The *MontiCore Data Explorer (MontiDEx) code generator*, which is the developed MDD tool support.

1.2 Objectives and Contribution

The goal of this thesis can be summarized in the following research question:

How can a generative approach support effective and agile data structure prototyping and development of customizable data-centric applications by adequate languages, extensible and adaptable tools, and methods?

This thesis aims to improve efficiency in the development and data structure prototyping of data-centric applications by reducing the necessary manual implementation effort for repetitive implementation tasks. In particular, implementing a GUI, a data structure that ensures data consistency, a connection to a persistence infrastructure, and support for process automation. By providing suitable languages, code generators, and methods, the goal is to achieve effectiveness of data-centric application development in an agile development environment.

In addition, because such an approach is always accompanied by customization, adaptation, and extension concerns (cf. [SVC06]), we aim for mechanisms that allow manually-written customizations of the generated source code to improve application developer's (cf. Section 3.1.3) acceptance (cf. [KBR11]); and a modular and adaptable code generator to enable code generator reuse (cf. [ZR11b]).

The main contributions of this thesis can be summarized as follows:

- Provide a language family that consists of the CD4A ML to describe the structural properties of analysis models omitting technical details, and the ADJava ML for process modeling.
- A systematic CD4A-to-Java mapping that maps each CD4A language concept to Java source code, which represents a data structure that ensures data consistency. This mapping explicitly resolves semantic variation points[4] in CD4A models by suitable *defaults*.

  The mapping also aims to avoid inspection of the generated implementation code, when customizing it. Instead, only the knowledge of the mapping and the generated interfaces is required to customize generated source code.
- A method to implement a modular and customizable data-centric infrastructure that is extended by a CD4A-specific part to use the data structure and implement a data-centric application or data-centric application prototype. The data-centric infrastructure uses a three-layer architectural style [BHS07] (a presentation, application, and persistence layer), which is divided into a model-dependent and model-independent part (cf. [SVC06]). For each model-dependent part of a layer, we present a mapping of CD4A language concepts to Java source code.
- The *Extended Generation Gap*-Pattern, which is a design pattern based on the Generation Gap-Pattern [Vli98, Fow10], to support customization of generated

[4] A semantic variation point is "a point of variation in the semantics of the UML metamodel and provides an intentional degree of freedom" [CCS13].

source code. However, the Extended Generation Gap-Pattern only requires the knowledge of the generated interface rather than the implementation and omits processing the generated source code. Furthermore, because it separates generated and non-generated artifacts, application developers (cf. Section 3.1.3) only need to version and manage non-generated source code.

- An extensible and adaptable code generation approach that is an integration of transformation- and template-based code generation. The approach maps a CD4A model to a CD4Code model, which is a DSL developed in this thesis to represent the object-oriented structure of the generated Java source code (subsequently called: *intermediate representation (IR)*); uses transformations on the IR; and allows to attach templates defining target language specific implementation details. This code generation approach supports generator developers (cf. Section 3.1.3) in designing modular and reusable code generators (i.e., MDD tools), and facilitate adaptation by supporting manually-written transformations and templates.

- A method to implement an extension of the data-centric infrastructure for modeling and execution of processes. This extended data-centric infrastructure enables modelers (cf. Section 3.1.3) to create ADJava models to specify processes without knowing the technical details. Each created ADJava model is executed by an interpreter. Action implementations can be added via handcoded extensions by application developers. Alternatively, senior application developers (cf. Section 3.1.3) can enrich ADJava models with Java source code. In this case, we provide a code generator that generates executable Java source code to reduce the manual implementation effort of application developers.

- A method to implement a code generator that realizes the proposed mappings and integrated code generation approach but additionally facilitates configurability and adaptability to enable code generator reuse. The proposed method aims to guide generator developers in developing configurable and modular code generators.

- To efficiently apply the proposed concepts and tools, lightweight methods for MDP and MDD of data-centric applications as well as guidelines for customization, adaptation, and extension of the technical realization are presented.

1.3 Organization of the Thesis

In order to answer the research question and present the developed concepts and methods, this thesis is structured as follows:

Chapter 2 introduces the foundations of data-centric applications. Moreover, the main elements of MDD are explained and the MC language workbench and code gener-

ation framework, which offers tool support for language engineering and code generator development, is described. This chapter also points out related approaches for MDP and MDD of data-centric applications.

Chapter 3 presents the envisioned methods for MDP and MDD of data-centric applications. These methods show how to use the proposed approach and points out the different roles and their concerns. In addition, the main requirements for this thesis are emphasized and structured according to the following dimensions: general considerations, modeling requirements, code generator requirements, and generated product requirements.

Chapter 4 introduces the CD4A ML to specify the structured information managed by data-centric applications. Furthermore, this chapter presents the CD4Code ML as an extension of CD4A ML. The focus of this chapter are modelers of data-centric applications and data-centric application prototypes as well as code generator developers using the proposed code generation approach.

Chapter 5 addresses generator developers and describes the mapping of the CD4A ML to Java source code, which represents the data structure. This chapter also highlights how data consistency is regarded by the mapping.

Chapter 6 presents the Extended Generation Gap-Pattern to customize the generated source code. This chapter, additionally, points out hot spots as an essential element for customization of generated source code.

Chapter 7 introduces the mapping of the CD4A ML to a data-centric infrastructure including a presentation, an application, and a persistence layer. In addition, a method to realize a generic persistence infrastructure for data-centric applications is described.

Chapter 8 describes the integration of transformation- and template-based code generation to provide a flexible and adaptable code generation approach. This code generation approach facilitates manually-written adaptations and extensions and is essential to support code generator reuse.

Chapter 9 presents the MontiDEx code generator that implements the proposed code generation approach and the mapping to generate data-centric applications.

Chapter 10 presents the extended data-centric application infrastructure to specify and execute ADJava models. This chapter shows a case example of extending the MontiDEx code generator via the provided customization and adaptation approaches.

Chapter 11 presents case examples in which the MontiDEx framework has been evaluated in the MDP and MDD of data-centric applications.

Chapter 12 concludes this thesis and presents the thesis' results. In addition, shortcomings and potential future research challenges are highlighted.

1.4 Related Own Publications

In the context of this thesis, different research publications have been contributed. They cover varying aspects of this thesis as pointed out subsequently.

- *Model Evolution*: Since the proposed approach highly relies on models, it is essential to address upcoming concerns such as model evolution, which focuses on the continuous change of models over time. A conceptual model for model evolution has been created [GLRR13, RGLR13, GLRR15].
- *Code Generator Product Lines*: The methods and concepts of extensibility and adaptability of code generators enable product line engineering. Code generator product lines essentially differ from common software product lines and as such are currently a research topic. Essential requirements for a code generator to support code generator product lines are pointed out [RR15]. The configuration and adaptation approaches developed in the course of this thesis (cf. Chapter 9) have been applied to different code generators [GMR+16, Mül17].
- *Code Generator Development and Testing*: Methods for developing and testing code generators are essential assets to enable their reusability and maintainability. An approach to develop composable code generators, which is based on explicit interface definitions, has been proposed [RRRW15]. This approach has been applied to the MontiDEx code generator to enable a modular code generator design [MSNRR15a]. Finally, an approach to support testing of such modular code generators in early stages of development has been presented [KLM+16].
- *Generation of Data-Centric Applications*: The MontiDEx code generator and the manual extension for developing data-centric applications have been demonstrated in [MSNRR15c].
- *Handcoded Extensions*: With the chosen MDD approach that is based on defaults and assumptions about the generated product, methods to mix generated and non-generated source code have been exploited to enable manually-written extensions of the generated products. An overview of existing approaches has been given and a new approach for handcoded extensions, which is used throughout this thesis, has been proposed [MSNRR15b, GHK+15a, GHK+15b].

Chapter 2

Foundations: Model-Driven Development and Data-Centric Application

While in Chapter 1 the need to reduce the manual implementation effort in the development and prototyping of data-centric applications is motivated, this chapter explains the concepts, methods, and tools on which this thesis is founded. In particular, MDD is introduced in Section 2.1. The MC framework for DSL and code generator development, which provides tool support for MDD, is described in Section 2.2. Afterwards, a definition of data-centric applications, which is used throughout this thesis, is given in Section 2.3. Finally, related approaches for MDP and MDD of data-centric applications are discussed in Section 2.4.

2.1 Model-Driven Development

Model-driven development (MDD) is a software development process that naturally resulted from the trend of abstraction and automation in software development such as computer aided software engineering (CASE) [Cas85]. Abstraction provides an approach to tackle the increasing complexity of software systems [BCW12], which is mainly caused by the high-level abstraction used by domain experts and the required low-level abstraction provided by *GPLs* (cf. [FR07]). This gap between high-level and low-level abstraction is bridged by automation [CCF+15], which systematically transforms the high-level abstraction to low-level abstraction (executable source code).

The high-level abstraction written down in textual or graphical notion is called *model*. Its meaning is not clearly defined (cf. [MFBC12]) but it is commonly accepted that a model is characterized by three characteristics [CCF+15]: there is an original that is modeled; the model is an abstraction of the original; and with respect to the original the model has a purpose. In this thesis, the following definition of a model is used:

Definition 1 (Model)**.** *"A model is an abstraction of a (real or language based) [software] system allowing predictions or inferences to be made." [Küh06].*

From this definition, we can conclude that a model describes a software system or a part of it. In general, two types of models are distinguished (cf. [FHR08]). *Descriptive*

models describe an original, e.g., existing software system, to better understand the original. *Prescriptive models* describe an original that is created from the model.

From this understanding of a model, MDD describes "the notion that we can construct a model of a system that we can then transform into the real thing." [MCF03]. MDD's defining characteristic is that models are lifted from documentary purpose and become the fundamental artifacts in the development process [AK03, MCF03]. They raise the level of abstraction to a description of relevant properties only. Hence, models "are considered equal to code" [SVC06]. Among others, this use of models separates MDD from model-based engineering and model-driven engineering (cf. [BCW12]). Nevertheless, the mentality that "everything is a model" [Béz05] is common to all processes (cf. [RdS15]). However, MDD does not describe when to use which model, because MDD is process-agnostic [BCW12].

MDD's main benefits are summarized as follows:

- MDD reduces the time needed to develop software, increases quality of software, and is platform independent [SVC06, Kul16].
- The communication between stakeholders is more effective and productivity is increased due to automation [BCW12, MGS+13].

Multiple domains including automotive [HRR12], cloud-based applications [NPR13], and robotics [RRW14, Wor16] have applied MDD. However, several challenges have been identified when introducing MDD in a software project. First, agile software development and MDD cannot easily be combined and require a combination of agile and plan-driven methodologies [KBR11]. Second, with the crucial role of models their quality, tooling for creating, managing, and refactoring models are of high importance (cf. [Ken02, CHN12]).

2.1.1 Domain-Specific (Modeling) Language

A model is not usable without means of writing it down. This can be done using a *general purpose modeling language (GPML)*, which is not tailored to a particular domain; or a *domain-specific language (DSL)*, which is a language that uses concepts from a domain. A DSL is defined as follows:

Definition 2 (Domain-Specific Language (DSL))**.** *A DSL is a "language that is specifically dedicated to a domain of interest" [CCF+15].*

This definition leaves out the clarification of the understanding of a language, which is a means for communication between stakeholders using a restricted amount of sentences [CCF+15]. Moreover, the terms *DSML* (short ML) and DSL as well as GPML and GPL are used interchangeably, because they cannot be separated [Loo17].

A DSL targets to bridge the gap between problem and solution space [CCF+15]. In contrast to GPLs, developers of DSLs favor restrictive design or even drop Turing-completeness. The border between DSL and GPL is vague but DSLs are typically smaller, usually declarative languages (cf. [VBD+13]).

2.2 MontiCore Language Workbench and Code Generation Framework

A DSL is defined by a concrete syntax, an abstract syntax, a set of context conditions, and semantics [HR00, HR04, MSN17]. The concrete syntax defines the language's representation using a textual or graphical notion [KRV07, GKR+07, MSN17]. The abstract syntax describes the syntactic structure of a model neglecting semantically irrelevant elements, i.e., syntactic sugar. In MC, the abstract syntax consists of a tree-like representation of the model, namely *abstract syntax tree (AST)*, and a *symbol table (ST)*, which stores additional information for model elements [MSN17]. In addition, context conditions are predicates defined with respect to the abstract syntax to determine the language's consistency [Sch12, MSN17]. Finally, the semantics define the language's meaning and can be specified in different ways as described in [MSN17].

A *language workbench* is a tool that enables agile language engineering of DSLs and provides tools to analyze, manipulate, and transform them. In contrast to existing language workbenches existing (many of them are listed in the Language Workbench Challenge [EvdSV+13]), the MC language workbench and code generation framework is lightweight and highly customizable [GKR+08, KRV08, Kra10, KRV10, Vö11, Sch12, Rei15, Hab15, Loo17, MSN17].

The MC language workbench operates as a generator generator, which uses a grammar definition to generate a generator for models defined by the grammar, or as a code generator that consumes models conforming to a grammar and produces source code.

MC's core architecture is shown in Figure 2.1. The input are models are processed by a `Parser` to create an AST (`AST`). `Control Scripts` using the `Workflow Execution` allow to choose and configure multiple workflows to be executed on the model, e.g., parsing, context conditions checks, or code generation. The `Template Engine` produces artifacts such as concrete source code or reports by using multiple `Templates` and the created `AST`. Technically, MC uses the FreeMarker template engine [www15a]. Finally, the `Workflow Execution`, the `Tempate Engine`, and the `AST` have access to common functionality, which is shared via the `Functional Library`.

2.2.1 MontiCore Grammar

MC uses its own EBNF-based DSL to define a grammar for a DSL. This grammar definition specifies the concrete and the abstract syntax of a DSL [KRV07, MSN17]. From

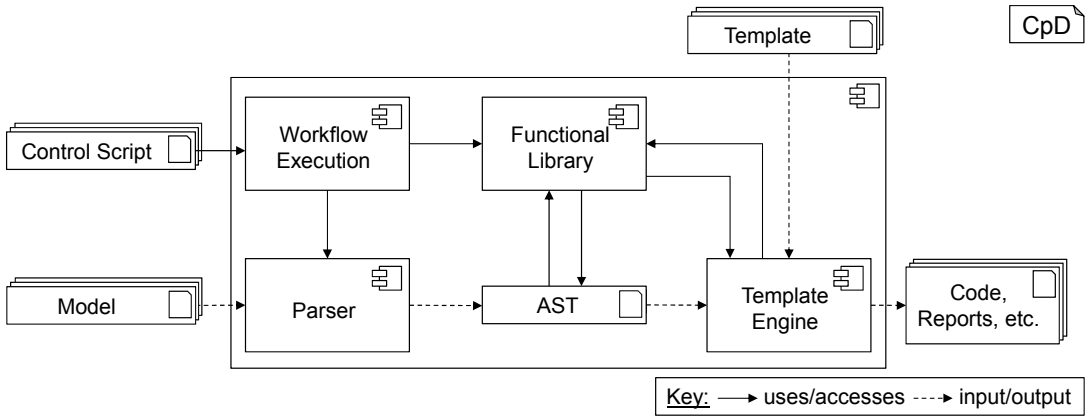


Figure 2.1: MC's code generation architecture for model processing and code generation based on [MSN17].

this grammar, MC generates an AST, a parser, visitors, and an infrastructure for context conditions for processing models conforming to the grammar. Subsequently, we briefly introduce the MC grammar DSL. Interested readers are advised to consider [MSN17] for a detailed description.

A MC grammar consists of *terminals*, *nonterminals*, and *productions*. Terminals are the atomic elements of a grammar and describe the concrete syntax of a language. Nonterminals are symbols that are defined by a production and can be replaced accordingly. Each production consists of a left-hand-side (LHS) and a right-hand-side (RHS), where the LHS defines a nonterminal and the RHS consists of lexicals, terminals, and nonterminals connected by predefined operators such as `"|"` (alternative), `"?"` (optional), `"*"` (arbitrary many), and `"+"` (at least one). Note that the RHS of productions can also be empty. In addition, MC supports inheritance, extension, and overwriting of productions.

For example, a MC grammar is shown in Listing 2.1. It describes the grammar for a lightweight UML CD DSL. Each grammar starts with the **`grammar`** keyword followed by a name (l.1). Terminals are defined between quotation marks, e.g., `"classdiagram"` in l.2. Terminals can also be marked as optional terminals, e.g., `["abstract"]?` in l.8. Nonterminals are defined on the LHS of a production, e.g., `CDDefinition` in l.2, and can also inherit from other productions by using the **`implements`** keyword. For example, in l.6 and l.8 the *interface production* in l.4 is implemented. An interface production introduce an interface in the AST and is defined with the **`interface`** keyword and a name. In addition, *lexical productions* can be defined by the **`token`** keyword and a name to define lexicals, e.g., `token Name = ('a'..'z')+`. MC has a set of predefined lexical productions (cf. [MSN17]).

MC supports additional advanced features such as *component grammars* to define incomplete grammars intended for extensions. Such grammars can be composed by

```
1 grammar SimpleCD {                                                    MCG
2   CDDefinition = "classdiagram" Name "{" CDType* "}";
3
4   interface CDType;
5
6   CDInterface implements CDType = "interface" Name;
7
8   CDClass implements CDType = ["abstract"]? "class" Name
9     ("extends" superclass:Name)?
10 }
```

Listing 2.1: An example of a MC grammar for a lightweight UML CD DSL.

inheritance or embedding and are essential to realize language inheritance and language embedding [LNPR+13, HLMSN+15a, HLMSN+15b, Wor16, Loo17, MSN17].

2.2.2 AST Generation from MontiCore Grammars

MC systematically synthesizes an AST from a MC grammar definition. In the remainder of this section, the systematic mapping of the MC grammar to an AST is described.

In general, terminals are not part of the generated AST. However, if a terminal is assigned with a name, e.g., `(type:"class" | type:"interface")`, then this terminal is mapped to a `String` AST attribute. Moreover, if a terminal is optional, e.g., `["abstract"]?`, the terminal is mapped to a `boolean` AST attribute.

Each nonterminal is mapped to an AST-class having the same name and a `AST` prefix, if it is on the LHS of a production. If the nonterminal is on the RHS, then it is mapped to a composition between the AST of the nonterminal on the LHS and the AST of the nonterminal on the RHS. Each lexical production is mapped to a `String` in the AST-class where it is used.

For example, Figure 2.2 illustrates the generated AST from the grammar specified in Listing 2.1. In this example, the `CDDefinition` nonterminal is mapped to the `ASTCDDefinition` AST-class having a `String` attribute for the `Name` lexical production. The same approach is used for the `CDClass` and `CDInterface` nonterminals, which are mapped to `ASTCDClass` and `ASTCDInterface` respectively. Moreover, the `["abstract"]?` terminal is mapped to the boolean value `isAbstract`. The interface production (l.4 in Listing 2.1) is mapped to the interface `ASTCDTypes`, which is implemented by the AST classes representing the implementing nonterminals, i.e., `ASTCDInterface` and `ASTCDClass`.

This example also shows how operators are mapped. The optional (`"?"`) operator is mapped to an optional variable as shown for `("extends" superclass:Name)?` (l.9 in Listing 2.1), which is mapped to an `Optional<String> superclass` variable.

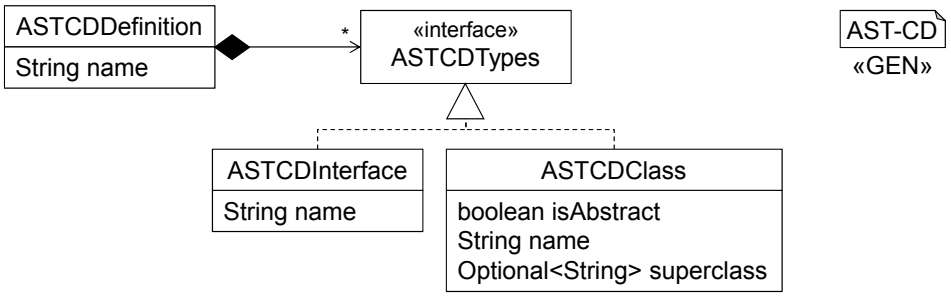


Figure 2.2: The generated AST from the MC grammar defined in Listing 2.1.

Note that the name given (`"superclass"`) is used for the AST variable name rather then the name of the lexical. Moreover, the arbitrary many (`"*"`) and the at least one (`"+"`) operators are mapped to a composition with `"*"` cardinality, e.g., the composition between `ASTCDDefinition` and `ASTCDTypes`.

2.2.3 Symbol Table

Besides the AST, MC provides an additional data structure, which allows to add additional model element information. Such an infrastructure is in particular necessary when dealing with names to enable efficient navigation between a name's definition and its use. Hence, names are the primary concept used in a *symbol table (ST)*, which is defined as follows:

Definition 3 (Symbol Table (ST))**.** *A ST is a data structure that maps names to essential information about a concept denoted by the name. [MSN17]*

In MC, STs are derived from grammar definitions in a systematic way. Hence, MC provides a code generator to generate a ST infrastructure. For a detailed description, interested readers are advised to consider [MSN17].

An element of a ST containing all essential information about a named model element is called a *symbol* and is defined as follows:

Definition 4 (Symbol)**.** *"A symbol definition (or short symbol) contains all essential information about a named model element. It has a specific kind depending on the model element it denotes. A symbol is defined exactly once." [MSN17]*

Each symbol has a kind that denotes the kind of the model element it refers to, e.g., class or enumeration. Depending on the symbol's kind, the additional information to be stored may vary. Each symbol has a visibility, which is the "region where the symbol is accessible through its name" [MSN17]. This visibility is controlled by access modifiers depending on the symbol's context, e.g., `private` and `protected` in Java. Moreover, a

symbol can be shadowed by other symbols, e.g., a local variable shadows a global variable in Java. Shadowing is within a model, whereas access modifiers determine access from other models.

Each symbol is associated with one *scope* to help defining language-specific STs:

Definition 5 (Scope)**.** *"A scope holds a collection of symbol definitions and impacts their visibility." [MSN17]*

In MC, scopes are tree-like structures attached to named elements in the ST. An example are local variables in Java, which are grouped together by the method in which they are defined. Scopes are distinguished into *shadowing scopes*, which may shadow names from their enclosing scope(s), and *visibility scopes*, which are all other scopes. By default, MC provides an *artifact scope*, which represents the scope of an artifact. It is the top scope of a model and represents a shadowing scope. Multiple artifact scopes are grouped by the *global scope*, which also forms the root of a scope tree.

2.2.4 Code Generation

The MC architecture (Figure 2.1 on page 12) can be separated into a *front-end*, which is concerned with language processing such as parsing, checking context conditions, and ST creation; and a *back-end*, which performs a model-to-text transformation of the AST to concrete source code.

Although no common understanding of a generator exists, the following definition combines its main elements (based on [Kra10, Sch12, RR15]):

Definition 6 (Generator)**.** *A generator is a software system that produces an implementation from a higher-level description of a (part of a) software system.*

A generator's defining characteristic is that it reduces the level of abstraction of the input by producing an implementation. Thereby, neither the input nor the target languages or how this model-to-text transformation is performed are predefined.

A special kind of a generator restricting the input to models (Section 2.1) and the target language to GPLs is the *code generator* (cf. [Jör13]), which is defined as follows:

Definition 7 (Code Generator)**.** *A code generator is a generator that produces an implementation using a GPL from a set of input models.*

This definition still contains code generators that hamper effective use in MDD. Hence, we define the following characteristics of a code generator: (i) a code generator always terminates, (ii) is deterministic, (iii) is not an interactive system, and (iv) generates at least one output artifact.

To perform the transformation from input to output, each abstract concept of the input language is mapped to concrete concepts of the target language (cf. [CE00, VBD+13]).

An overview of existing approaches for this transformation is shown in [Jör13, VBD+13]. Among them, template-based [Wac09, Kra10, Sch12, CT13] and transformation-based approaches [CE00, HKGV10, EBBG12, DREP12] are the most common ones used by modern code generation frameworks such as Xtext [Bet13], MPS [Cam14], Acceleo [www16a], and Spoofax [www16g].

Subsequently, template- and transformation-based code generation are described, because they form the foundation for the integrated code generation approach, which is proposed in this thesis and explained in Chapter 8.

Template-based Code Generation

Template-based code generation is based on templates as the primary development artifact. Each template consists of plain target language code and additional template language instructions. Such templates are processed by a template engine to generate source code conforming to the target language. In addition, a template may require input arguments. For example, in MC the AST of the input model is passed to each template such that it can be used by template language instructions.

An overview of template-based code generation based on [Kra10, Sch12] is shown in Figure 2.3. Assuming that an `AST` has been successfully created by a `Parser`, a `Template Engine` executes a template for each AST node type, which has previously been developed for this particular type of AST node. Such templates may call sub-templates or `Embedment Helpers`, which are plain Java objects aimed to reduce complexity of templates by outsourcing dedicated functionality from templates. When the template engine evaluates a template, it accesses the AST element to compute the result for template instructions. Afterwards, the result, which is target language-specific source code, is written into a file.

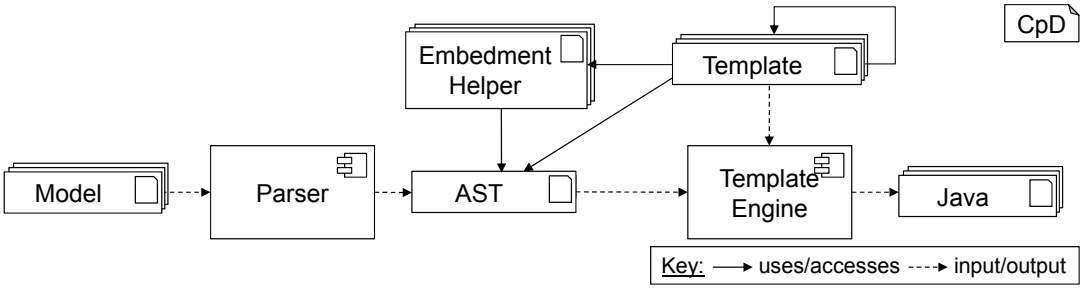


Figure 2.3: An overview of template-based code generation based on [Kra10, Sch12].

A benefit of template-based code generation is that target language code can directly be written into templates. This is especially necessary when generating GUIs, as identified in [MK09]. However, template-based code generation may result in unmaintainable

templates because target language source code is mixed with template instructions. This challenge is only partially addressed by research (e.g., [CT13]). Another drawback is that ensuring static correctness requires parsing the generated source code [HKGV10].

Transformation-based Code Generation

Transformation-based code generation is another type of code generation. Its primary development artifact are transformations, each of which manipulates either the AST of one particular language (*endogenous transformation*) [CH03, MCvG05]; or performs manipulations between ASTs of different languages (*exogenous transformation*) [MCvG05, MvG06, HKGV10]. Due to the varying understanding and classification of transformations (e.g., [CH06, Ste08, JBW+14]), we define a transformation as follows:

Definition 8 (Transformation)**.** *"A transformation is the automatic generation of a target model from a source model, according to a transformation definition. A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language." [KWB03]*

The overview in Figure 2.4 shows the use of the exogenous transformations for code generation. After processing a model, the `AST` of the input model can be used to execute endogenous transformations. Such transformations can, e.g., be employed to reduce language concepts (*normalization* in [HKGV10]). To generate concrete source code, exogenous transformations are used to create an AST of the target language, e.g., Java source code. This is typically done by a `Transformation Engine`, e.g., [Wei12]. Finally, the textual output is produced by systematically printing the transformed AST of the target language, e.g., by using a pretty printer (*unparsing* in [Kla06]).

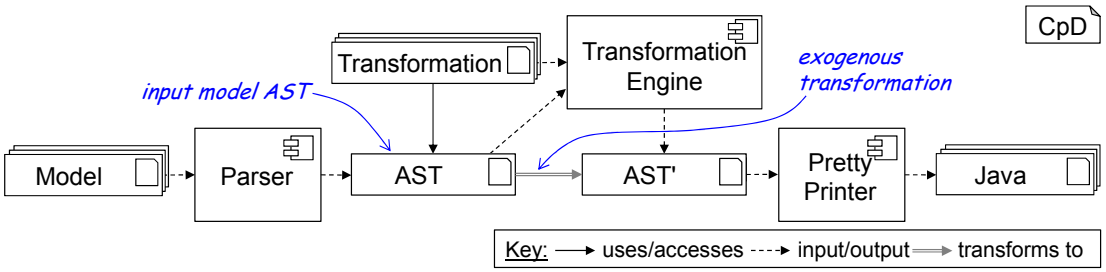


Figure 2.4: An overview of transformation-based code generation using exogenous transformations to create the target language's AST.

A benefit of transformation-based approaches is the replacement of calculations on and traversals of the input model by pattern matching, i.e., search for the input language constructs for which a rule is defined (cf. Def. 8). In cases where domain-specific

transformations are employed, e.g., [Vis02, BW07, RW11, Wei12], the concrete target language syntax can be used to describe the transformation [HKGV10, HRW15]. In addition, because transformations produce a manipulated AST, well-formedness checks can be applied to the input and output AST.

Nevertheless, transformation-based approaches require an abstract representation of the input and output model [HKGV10], and a corresponding transformation language. Moreover, the output AST contains every detail of the source code to be generated, which may not always be wanted, e.g., method implementations.

2.3 Data-Centric Applications

Data-centric application consider management of structured and consistent information as their primary focus. In data-centric applications, the managed structured information and the application's behavior are separated in the application's design. In particular, the managed structured information represents the application's state and encapsulates the application behavior. In this thesis, a data-centric application is defined as:

Definition 9 (Data-Centric Application)**.** *A data-centric application is a software system that is build around structured and consistent information to provide means to display, manage, and manipulate such information persistently.*

This definition encloses a variety of software systems. In this thesis, a data-centric application is restricted to a lightweight client application providing a GUI, a client-side representation of the managed data, a connection to a persistence infrastructure, and means to support process automation. A data-centric applications can be used as a front-end for InfoSyss or *enterprise applications* [Loo17].

Unlike software systems that are built around unstructured data, e.g., plain text documents, a data-centric application understands the managed structured information to access and manipulate it. Hence, the strengths of a data-centric approach is that information is structurally broken down into essential parts of the application. Each part is identifiable in the software system's architecture. In addition, because all relevant data is persisted, data-centric applications can be stopped and restarted from the stored data.

Technically, a data-centric application is realized by a *data-centric infrastructure*:

Definition 10 (Data-Centric Infrastructure)**.** *A data-centric infrastructure is a technical realization of a data-centric application that extends a data structure with management capabilities accessible via a GUI.*

In this thesis, management capabilities refer to the (S)CRUD-operations on the managed data and a connection to a persistence infrastructure. Such a data-centric infrastructure is independent of any programming language or architecture. However, an architecture that has proven to be beneficial for such kind of software systems is the layered architecture, which is explained subsequently.

2.3.1 Layered Architecture

The layered architecture is an architectural pattern that decomposes the overall complexity of a software system into multiple stacked interacting layers, each of which represents an aspect of the overall software system [BHS07]. Layers represent a unidirectional "allowed-to-use relation" [BCK12] meaning that an upper layer is allowed to use functionality of lower layers. It is, however, possible to break this unidirectional relation in some cases [Sta09, BCK12], e.g., performance reasons.

A common layering for data-centric application consists of a *presentation layer*, an *application layer*, and a *persistence layer* (cf. [Sta09, BCK12]), each of which serves a designated purpose:

Presentation layer (top layer) is concerned with presenting the data and providing functionality to the end user to manage, search, and filter the data.

Application layer (middle layer) is responsible for providing business logic that is defined relative to the managed data. It represents the domain elements of interest.

Persistence layer (bottom layer) is responsible to store data in a consistent and permanent way. In this thesis, it is assumed that a persistence infrastructure is provided, as further explained in Section 7.4.

2.4 Related MDP and MDD Approaches

In this thesis, related work is discussed in each chapter whenever appropriate. In this section, approaches for MDP and MDD that are similar to the approach proposed in this thesis are presented. In this regard, similarities and differences are explained. However, the majority of existing approaches for MDP of data-centric applications focuses on GUI prototyping (an overview of existing approaches is given in [ABY14]). Hence, from such approaches only those are described that provide management of data structures on the client. Similar holds for rich internet applications that are omitted, because focus of this thesis is on client-side applications. For the same reason, only the most common approaches for MDD of InfoSyss that address similar concerns are regarded. Existing approaches for enterprise information system and web information system development are listed in [Loo17] and [Rei15] respectively. Furthermore, a general overview of existing agile methods to develop different types of software systems is shown in [Alf16].

2.4.1 Related MDP Approaches for Data-Centric Applications

An approach targeting rapid prototype generation of fully functional data-centric applications has been proposed in [MP03, MP04]. The approach uses a UML profile for business applications to describe the data structure [PMDM11]. Models conforming to

this UML profile are used to generate source code that extends a generic application, and generate a database schema. The generic application is an application that comprises an union of coarse-grained components (subsystems are build by creating sets of associated components). In addition, the generic application provides generic forms, which are extended to build model-specific forms. Each generic form is dynamically adapted at run-time. Moreover, the generated application provides CRUD operations and more complex manipulations for the modeled data structure. A code generator used in this approach consists of a model analyzer, which imports models from a repository and generates a specification for a HCI; a forms generator, which implements a GUI for previewing and editing the application specification produced by the model analyzer; a DMP generator, which supports CRUD operations and more complex operations modeled via a modeling tool, for complex data manipulation procedures; and a documentation generator, which generates HTML and RTF documentation. Adaptations of the generated application are realized within the forms generator, which parses the generated code and detects changes. In contrast, the approach proposed in this thesis uses a lightweight data-centric infrastructure that has to be extended by generated source code rather than a generic application. Moreover, in our approach complex CRUD have to be manually added to the generated source code. The approach proposes a method for MDP, which servers as the foundation for the developed method in this thesis. The proposed MDP method consists of: requirements elicitation, prototype design, prototype construction, prototype evaluation, and exploitation.

A different approach targets to introduce object-oriented analysis (OOA) techniques in an educational context [MSHL06]. It addresses the development of a modeling and implementation approach that is scalable to real life systems. Hence, from a conceptual domain model, which is a variant of a UML CD model; a proprietary object-event table, which is a mapping between events and class in the UML CD; and a group of finite state machines, prototype applications are generated. Each prototype application consists of a J2EE application with a Java Swing GUI using a three layered application with a GUI, event handling, and a persistence layer. The mapping from the abstract models to the prototype is done by a platform-independent-model (PIM) to platform-specific-model (PSM) transformation. This approach shares a similar architecture as the approach proposed in this thesis. However, it has the following limitations:

- Lacking a mapping of hierarchies in the conceptual domain model to source code.
- The conceptual domain model is restricted to associations with cardinality 1-to-1 and 1-to-many.
- The generated source code does not ensure data consistency.

Another approach for generating prototypes from unified modeling language (UML) models of system requirements has been proposed [LLHL05, LL08]. A UML model of

system requirements is considered as a UML use case diagram and a conceptual class model. A UML use case diagram is specified by pairs of pre- and postconditions. A conceptual model consists of classes with attributes and associations. Both are textually described using XML. From these models, this approach targets iterative and incremental development of evolutionary and throw-away prototypes to validate requirements. From such models a system entity object database (SEOD) with CRUD functionality for each class and association is generated. In addition, a Use-Case Handler is generated, which transforms the pre- and postconditions into executable CRUD operations on the SEOD and handles UML use case diagram execution. Finally, a basic GUI prototype containing buttons for each use case is generated. In contrast to the approach proposed in this thesis, this approach targets prototyping based on use cases and is limited in the generated GUI to perform all CRUD operations independent of use cases. Furthermore, customizability and adaptability are not explicitly addressed.

An approach to generate a functional prototype from a UML CD and a Statechart (SC) has been presented in [FBLS12, Let14a, Let14b, GAL15]. It is based on the Umple modeling tool and programming language family, which consists of a DSL for UML CD and a DSL for SC. The language family supports integration of target language source code such as Java, PHP, C++, or Ruby. Hence, handwritten extensions of the generated source code are located in the same artifact as the model itself. A goal of Umple is to support reuse at various levels of abstraction [FBL10]. Therefore, Umple supports mixin because they are "useful for prototype development because different features can be independently described in Umple and added one-by-one to a base model to explore their ramifications" [FBLS12]. For example, such mixins allow to define a class in the UML CD DSL twice describing different aspects such as method implementations. A code generator is used to generate a prototype from the Umple class diagram and Umple SC. The generated prototype respects *semantic rules*, i.e., referential integrity and multiplicity constraints at run-time. Moreover, the generated prototype is web-based and supports CRUD operations, shows all attributes, and allows to follow associations. The mapping proposed in this approach to ensure data consistency is used as the foundation for the mapping proposed in this thesis. However, the mapping proposed in this thesis additionally ensures that all Java variables generated for CD4A attributes are not assigned a `null`-value. Moreover, the mapping developed in this thesis ensures that underspecification in CD4A models, e.g., association's navigation direction and association's cardinality, are resolved by defaults to enable use in early development stages. Another distinction to this approach is the support for methods in the DSL for UML CD. Methods are not supported in CD4A due to the focus on analysis models (cf. Section 4.1).

2.4.2 Related MDD Approaches for Data-Centric Applications

The JANUS application development environment [BHKN96] proposes a MDD approach for client-side data-centric applications with a client-server architecture. Such a data-

centric application is generated from one OOA model, which is a language family consisting of three languages described via the Janus Definition Language. In particular, a UML CD model is used to represent the application's *object model* of the problem domain. A *GUI knowledge model* is employed to describe different aspects of the GUI such as layout strategies. It also maps the OOA model to user interface (UI) elements. Finally, a *meta-model model*, which is primarily used for code generation, to merge all other models has to be created. A code generator - which does not explicitly address extensibility and adaptability, in contrast to the approach proposed in this thesis - generates a software system from these models that consists of a GUI, an application layer, a persistence layer, a database schema, and further services such as printing, sorting, and deleting. However, the generated software system represents a code frame that has to be completed by the developer. For example, methods defined in the OOA model have to be implemented. This contrasts to the purpose of analysis models used in this thesis, which do not contain methods. The generated application layer connects GUI and persistence layer, which provides read and write operations for attributes of the OOA model. The GUI layer offers default values that can be overridden and uses the following mapping from OOA models:

- Map attributes to editable UI fields with label. Supported fields are: edit field, text field, combo boxes, drop-down combo boxes, list box, drop-down list box, check boxes, radio buttons and tables.
- Map associations and aggregations to editable lists.

With this mapping a list and edit view is provided for each OOA model type. Each edit view provides input validation. However, the GUI is not completely generated but uses a JANUS Application Framework, which is similar to the run-time environment (RTE) used by the data-centric infrastructure (cf. Section 7.1.3). The mapping from OOA models to a GUI proposed in this approach forms the foundation for the mapping developed in this thesis.

A MDD approach for large scale enterprise applications development has been proposed in [KVR02, Kul16]. It employs multiple DSLs to describe every aspect of the generated software system independent of the target language and platform. To capture business entities and their relations, an extended DSL for UML CDs is used. It is extended to capture additional architectural design decisions, which are regarded during code generation. For example, classes to be displayed in the generated GUI can be marked, or classes to be persisted can be denoted as well. Such a design decision for the DSL is contrary to the CD4A ML, which is intentionally kept simple to avoid mix of design decisions and analysis models. In addition to a DSL for UML CDs, DSLs for UML use case diagrams and UML ADs are used to describe process flows. Finally, to describe business logic independent of target languages, a the Q++ DSL is provided. In the approach proposed in this thesis, business logic has to be manually added via

customizations of the generated source code (cf. Chapter 6). The variety of models used in this approach is kept consistent via interconnections between the different DSLs. A code generator processes all models and generates a business application using a layered architecture consisting of an application layer, GUI layer, and database layer. However, the variety of the used DSLs and the developed software systems have shown the following major challenges:

- The approach has a steep learning curve. In this thesis, this is addressed introducing customization of the generated source code on generated-source-code-level to keep familiarity of application developers.
- New requirements may result in implementation of new code generators. In this thesis, this challenge is addressed by providing extension and adaptation mechanisms for code generators.
- Maintenance of the languages and code generators. This challenge is addressed in this thesis by reducing the number of languages and only one code generator.

To achieve scalability, components (building blocks) that specify classes, operations, and queries have been proposed [KR08, BK10]. Each building block defines variation points and can be composed with multiple others to create an enterprise application. In contrast, this thesis does not address large scale enterprise application development but uses modularity on the code generator level (MontiDEx modules described in Section 9.2) and data-centric application layer (cf. Section 7.1.3).

Another approach for MDD of data-centric applications is a generative framework for the development of CRUD-based Linux desktop applications [JC15]. The main goal of this approach is the contribution of a design and an implementation of a framework prototype for simplifying the development of desktop applications. To achieve this goal, this approach proposes the use of three models. First, a *domain model* describes the structure of the managed data. It is represented as a list of entity types, where every entity type has either a value type or a relationship, which is restricted in cardinality to 1:1 and 1:n cardinality. In contrast, the CD4A model does not have such a restriction and supports all associations with arbitrary combinations of `[1]`, `[0..1]`, `[1..*]`, and `[*]` cardinality. Second, a *wrapper model* that wraps entities to be displayed in the GUI by "flattening" each entity, i.e., resolve associations. In addition, the wrapper model allows to adapt data types and assign names to elements to be displayed in the GUI. Third, a *UI model* to describe the UI in a YAML-like way. Whereas UI models conform to a DSL, the domain model and the wrapper model are embedded in the Vala DSLs. From these models, a code generator generates a database schema script and a Model-View-ViewModel pattern-based for the GUI. A similar variant of the Model-View-Presenter-Pattern is generate in the approach used in this thesis (cf. Section 7.3). Moreover, the generated application uses data access objects (DAOs) and Builders to

manage instances. In our approach, a DAO and a Builder is generated for each CD4A class. Additionally, loading strategies in the generated application can be influences by bindings of the domain elements and UI elements. In our approach, a default loading strategy is generated (cf. Section 7.5.1) but can be adapted via handcoded extensions. In addition, the code generation process can be configured by configuration options, additional stereotypes in the domain model, and manually-written extensions of the generated source code. The generated source code is compiled using an adapted Vala compiler, which reportedly was the cause for many challenges in this approach. In contrast, in the approach proposed in this thesis configuration options are script-based facilitating to configure every aspect of language processing and code generation (cf. Section 9.3). Furthermore, additional black-box adaptation mechanisms to extend and adapt the code generator are provided (cf. Chapter 8).

In [Sol10] an approach to generate CRUD-based JavaFX GUIs from UML CDs models is described. To achieve modularization and customization of the code generator the approach proposed a template-based code generation approach based on semantic graphs, which can be seen as a IR (cf. Section 8.2.2). In contrast to our approach, black-box adaptation is not addressed. The generated data-centric application has a layered architecture, which uses separation of source code and different design patterns to achieve modularization of the generated source code. A similar goal is targeted in this thesis (cf. Section 7.1.3). However, the generated data-centric application only generates a fully functional GUI and provides means to manually implement CRUD operations. In contrast, in our approach a completely working data-centric application is generated. Moreover, underspecification in the input model is not resolved hampering generation of fully executable applications. In our approach, underspecification is explicitly addressed and resolved (cf. Section 9.1).

An extension of the MDP approach for data-centric applications [MP03] has been proposed to support development of business applications [MFM+13]. The approach aims to close the mental gap between mapping user requirements to UML models. Therefore, the *application model* is integrated in the overall development process. The application model is extended with additional design guidelines and a DSL to design and describe complex UIs form designs. On contrary, our proposed approach uses a default mapping that has to be manually extended. A code generator generates a GUI, business logic, and a database schema. However, manual interventions may be required to transform the persistence model to the database schema. In our approach, this is not required, because the persistence infrastructure is assumed to be generic and model-independent. In addition, the same authors have also explored design of a DSL for generation of database applications (can be understood as data-centric applications by the definition in Section 2.3) with CRUD operations [DMPT10]. The generated application, provides validation support but omits data consistency. For customization concerns a protected regions approach is used in the generated source code.

To teach students MDD, a lightweight approach to generate CRUD-based applications has been presented [PBCN14, PBCN15]. It aims to address the following concerns: data entry, data validation, data persistence, and data presentation in each CRUD application. This is achieved with four DSLs. First, an entity DSL to describe the data structure of the data to be managed. Second, a constraint DSL is introduced to describe different constraints on the entity model. Third, an entity DSL with references, which extends the entity DSL with additional references between entities. Finally, a UI specification is added to define the GUI. From these models a DAO implementation, a service interface, and a console-based desktop UI is generated. The generated source code depends on a RTE, which contains non-generated source code. In addition, the generated application can be extended with additional handwritten source code. Our approach shares similarities. In particular, a RTE is used, DAO implementations are generated, and support for handcoded extensions is provided. However, this approach uses a service-oriented architecture and provides a limited console-based UI. Furthermore, it does not explicitly address data consistency as well as extensions and adaptations of the code generator.

A further MDD approach is presented in [REM15]. The authors propose a lightweight approach and methodology for the generative development of JavaFX GUIs using the Model-View-Presenter (MVP)-Pattern. Rather than describing a domain model, the authors aim for the design of a GUI via a meta-model. Each instance of this meta-model is transformed to an instance of a second meta-model for the targeted application. Transformations are used to generated source code. Similar to the approach proposed in this thesis, the authors use an IR for code generation. However, neither customization concerns of the generated source code nor MDD tool reuse is addressed.

In [SPHV10], the authors present a rapid MDD approach for UI development in large enterprise projects. The approach proposes a lightweight method consisting of (i) creating annotations, (ii) defining layout and reference domain model, (iii) running the application, and (iv) customizing the generated application. The input is a UI container model, which holds information about the UI container structure and the reference to a domain model. Hence, the second input is a domain model, which is assumed to be non changing for a particular domain, which is in contrast to our approach. The authors propose tooling to rapidly create the UI container model. Rather then generating executable source code from these models, the authors propose an interpretation-based approach. To customize and influence the interpretation, model elements can be annotated using a predefined set of annotations. This annotation-based customization is extended to per-user annotations to support individual annotations for different users. In contrast to the approach proposed in this thesis, this approach aims to UI development only, addresses different customization mechanisms, and does not represent the managed data sufficiently to support client-side data-centric applications.

The approach presented in [Sub15] addresses end users rather than developers for the development of InfoSys. It introduces a novel model-driven engineering (MDE) approach, called BUILD, that is based on transformations that allow to predict and

provide new knowledge at each transformation step. Hence, from a lightweight modeling language (μML), which is used by end users to specify the targeted InfoSyss, a four-phased process is started. It starts with a requirements sketching phase, followed by an analysis phase. Then, a design phase and a code generation phase. In this process different models are used to specify the overall software system. From these models a GUI, the data structure, and a database is generated. The approach presented in this thesis, primarily targets application developers rather than end users. Moreover, only one model to describe the data structure is used and defaults are assumed to generate the overall software system. In contrast to this approach, the approach proposed in this thesis addresses customization and adaptation concerns.

Finally, a lightweight and agile model-based development (MBD) approach that is applicable for many areas of applications is explained in [Lan16]. The approach proposed in this thesis uses a similar lightweight and agile method. Similarities of our proposed approach and this approach are the framework-like use, lightweight and agile methods, and common mappings. Hence, in the remainder of this thesis, this work is compared whenever appropriate. However, this approach uses dedicated DSLs for class diagram, use case diagram, and UML object constraint language (UML OCL) definition. The primary goal is to completely specify the targeted software system and use transformations to generate executable source code. Rapid changes are addressed through propagation via code generation rather then handcoded extensions as introduced by this thesis.

Chapter 3

Requirements for the Envisioned Methodology

After Chapter 1 outlined that MDD improves prototyping and development of data-centric applications, and the foundations of this thesis have been explained in Chapter 2, this chapter gives an overview of the envisioned methods by example. The goal of this chapter is to demonstrate how MDD can successfully support prototyping and development of data-centric applications, and reveal requirements, roles, and tasks that have to be addressed by tools and methods. This chapter also provides the methodological guidance for this thesis.

This chapter is structured as follows. First, a typical scenario of a software company developing a new data-centric system is presented in Section 3.1. Afterwards, the primary high-level requirements for this thesis are described in Section 3.2. Finally, the envisioned methods for MDP and MDD of data-centric applications are explained in Section 3.3.

3.1 Typical Scenario for Generative Development of a Data-Centric Application

In our scenario, a software company aims to replace their current applicant management system due to unsatisfied needs in the managed information referring the application process. However, the diversity and depth of all requirements is vague and mostly unknown. Hence, before developing a new software system, the company aims to use a prototyping approach to elucidate all requirements.

In general, *prototyping* is the idea to develop a first version of the software system that is used to test certain aspects of the targeted software system [LL13]. The main goal of prototyping is "to get a feel of how the full system will behave" [FBLS12]. In contrast to classical prototyping (e.g., [Som10, LL13]), where prototypes are developed manually, the company decides on a MDP approach to automatically generate prototypes from abstract descriptions (e.g., [FBY08, MBR08, BDLD11, FBLS12]). Rather than prototyping the GUI, their goal is to identify the data structure, which represents the data to be managed by the new applicant management system. Nevertheless, a GUI providing management

functionality for the data structure is needed to enable a hands-on prototype that can be experienced by *end users*.

3.1.1 Model-Driven Prototyping of Data Structures

From the currently used applicant management system, a preliminary understanding of the data to be managed is developed by a *modeler*. The UML CD shown in Figure 3.1 represents the data currently managed. It contains a `Person` having a first name, a surname, an email address, and an age. Each `Person` has a gender. In addition, at least one `Address`, which has a street, a number, and is located in a city, is associated with a person. To manage applications, each `Person` can apply for a `JobPosition` by creating an `Application`, which contains the date of receipt. Each `JobPosition` has a description, a begin date, and is associated with a `Company`. Each `Company` has a name, a number of employees, and may have multiple open job positions.

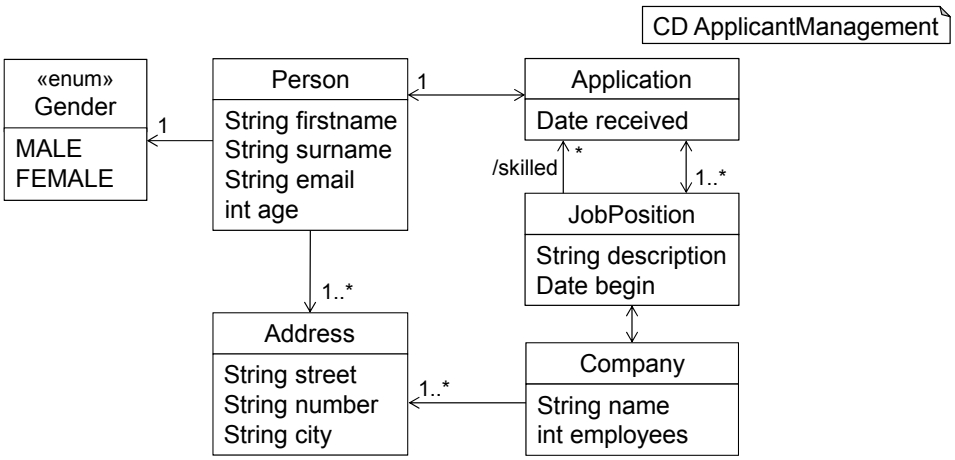


Figure 3.1: A UML CD for a lightweight applicant management system.

From this initial task in the development of a prototype, a GUI has to be developed and the UML CD model has be mapped to executable source code. In MDP, these tasks are performed by a code generator. Hence, the *application developer* executes the code generator using a CD4A model, which realizes the UML CD model in Figure 3.1, as input. However, before executing the code generator, he has to choose the parts of the data-centric applications that are generated by either selecting an existing code generator configuration or create a new one. In this scenario, the application developer chooses the configuration to generate all three layers of a data-centric application.

The functionality of the prototype can be realized either horizontally (all functionality is provided but not realized in great detail) or vertically (only few functionality is provided but realized in great detail) [LL13]. The goal of this thesis is to generate

a fully functional prototype that is horizontally and vertically complete for this kind of software systems, i.e., a GUI offering (S)CRUD (Search, Create, Read, Update, and Delete) functionality for the managed data structure is provided. As a result, these rich prototypes can be used for every type of prototypes including throwaway, evolutionary, and even incremental prototype [Som10].

The prototype generated from the UML CD model in Figure 3.1 is shown in Figure 3.2. It shows all elements of the data structure and provides means to manage them via a GUI. For certain inputs, validation support is provided to ensure that the user enters valid values. Moreover, Figure 3.2 also shows feedback on mandatory associations (`Address` list marked red) to show the constraints (i.e, mandatory associations) modeled in the UML CD model.

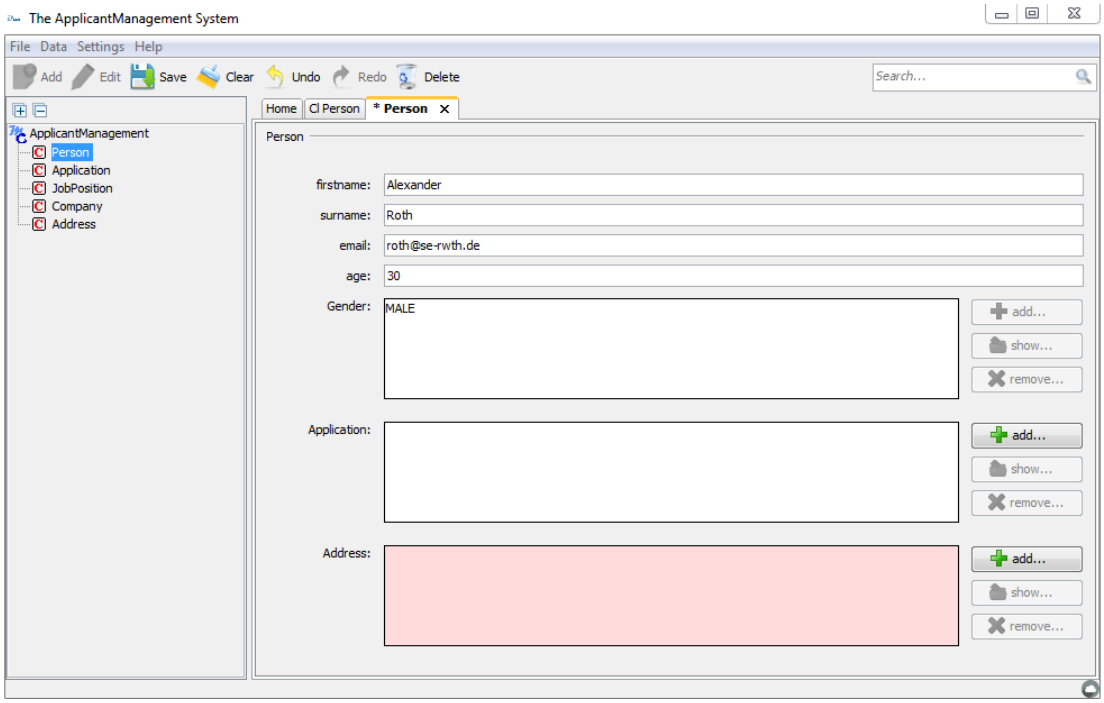


Figure 3.2: Generated prototype from the UML CD shown in Figure 3.1.

However, the generated prototype does not realize all requirements identified by the modeler. In particular, the derived association from `JobPosition` to `Application` in Figure 3.1, which links all skilled applicants to the job position. Therefore, the application developer customizes the generated source code using target language source code and adds the required implementation. This approach is necessary for all requirements that cannot be described in the abstract model.

The manually customized prototype is used by the modeler for testing purposes with the end user to identify further requirements. For example, the type of the company or if the job position requires the applicant to manage personnel. For each new requirement the model is adapted and the code generation is repeated. During this iterative and incremental development process, the manually-written customizations are preserved.

3.1.2 Model-Driven Development of Data-Centric Applications

After multiple prototyping iterations, the data structure has been identified. Using the elucidated requirements, the company aims to develop the new applicant management system. To reduce the development time and costs, the company's plan is to gradually replace their current system using an agile MDD approach. Hence, they aim to reuse both models and the code generator for their new software system. However, the developed software system should (i) provide different roles for users with varying rights, (ii) support local file storage, and (iii) provide a JavaFX GUI rather than a Java Swing GUI.

Therefore, the application developer sets up a new configuration for the code generator by only choosing the application and persistence layer of the data-centric application. The generated source code should serve as a framework-like part of the software system. Afterwards, he executes code generation and the Java source code is generated. Next, the application developer implements the GUI by extending the generated source code. While the GUI can be built on top of the generated framework, the serialization support requires adaptation of it. However, manually-adapting the generated source code is not practical, because multiple generated artifacts are involved and the company plans to reuse this functionality. Hence, a *generator developer* extends the code generator with this functionality by using the code generator's predefined extension points and adaptation mechanisms. Therefore, he provides a manually-written template and transformation that is added to the code generator configuration.

During the development of the software system, the *infrastructure provider* sets up an infrastructure such that managed data can be stored persistently. Therefore, he sets up an application server providing storage capabilities and role-based access control. This functionality is accessible via a service that realizes the methods required by the data-centric application's persistence layer. Afterwards, the application developer adapts the persistence layer to use the provided application server by setting the IP-address.

A further requirement of the company is automation of business processes in applicant management. Due to the changing regulations, their business processes are regularly adapted and are also different for each end user. Hence, business processes have to be adapted even after deploying the data-centric application.

In our scenario, the business process shown in the UML AD in Figure 3.3, which illustrates the process to handle new applications, should be automated. Each new application is checked whether the applicant is skilled for the job position or not. Only if the applicant is skilled, the manager invites the applicant to a meeting.

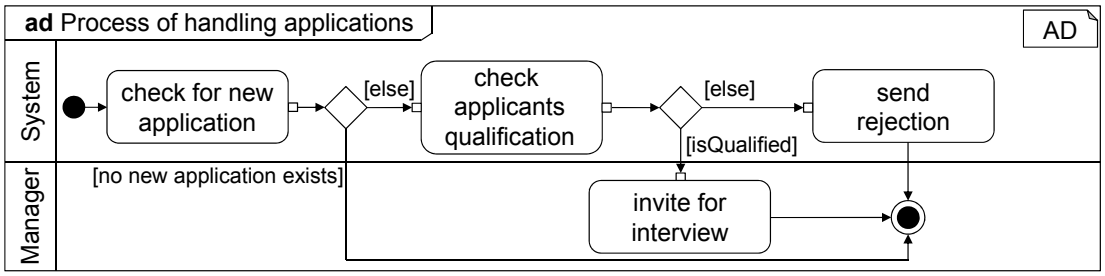


Figure 3.3: A UML AD of a process to manage new applicants.

For this process an ADJava model is created by a modeler, who is unaware of implementation details. A *senior application developer* enriches the ADJava model with Java source code implementing the technical details such as sending a rejection or checking if the applicant is qualified. Afterwards, code generation is started by the senior application developer to generate executable source code.

A modeled process is executed by end users via the GUI shown in Figure 3.4, which provides actions (buttons at bottom left in Figure 3.4) to: execute a process (Run button), pause a process (Pause button), save a currently paused process (Save button), stop the currently executed process (Stop button), clear the log information (Clear button), and access to settings to set the debugging level and path to the ADJava model (Settings button). If adaptation of the process is needed, application developers can manually extend the product even after deployment, where a code generator is unavailable.

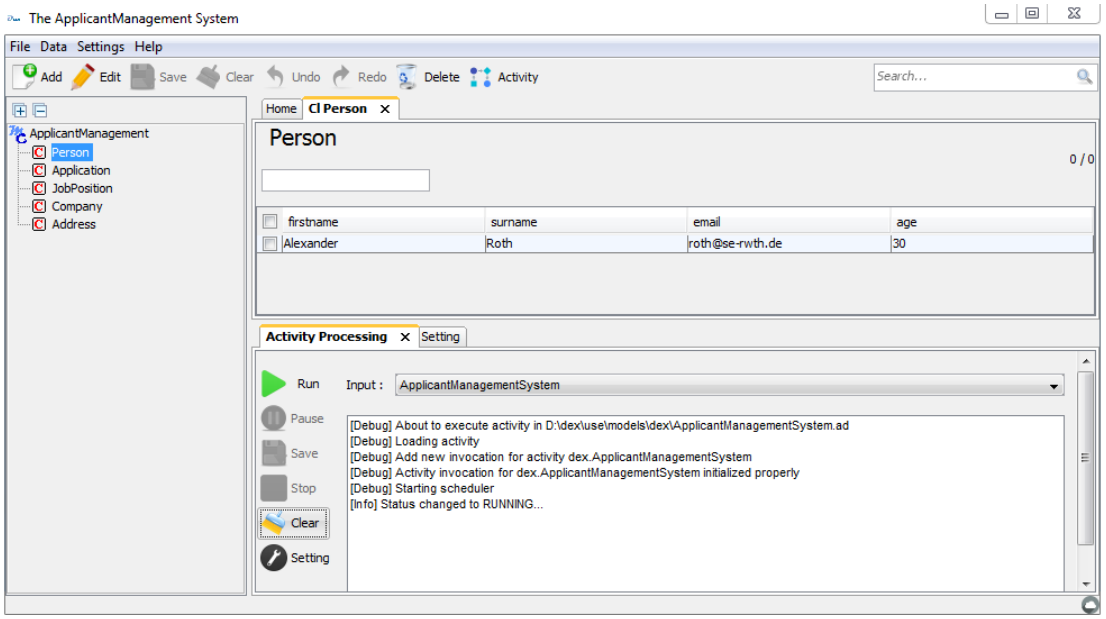


Figure 3.4: Integration of a GUI for process automation in a data-centric application.

3.1.3 Roles in the Development and Prototyping Process

Each software development project consists of a team that incorporates different roles, each of which describes the responsibilities of team members in the particular process [Som10]. The different roles have distinct but sometimes overlapping concerns, interests, and goals. Nevertheless, each of the role's needs have to be fulfilled in a successful software project.

In this thesis, it is assumed that the team is small as the project size is assumed to be small to medium sized projects. Subsequently, the varying roles and their characteristics based on the ideas of [Her13, LL13, Wor16] are identified.

Modeler. The modeler is one of the most important stakeholder in the proposed usage scenarios described in Section 3.1. The modeler is focused on identifying the end user's requirements and the domain itself [MP03]. This requires to deeply understand the end users' interests, concerns, and their daily work routine. Hence, knowledge in requirements engineering [BPKR09] or domain engineering [CE00] is a necessity. His main concern is to create models from the identified requirement (cf. [Kul16]). However, as the modeler is only concerned with the structural and behavioral modeling of the system, he is unaware of the technical realization.

This role can be further broken down into roles focusing on particular concerns, e.g., system designer, concern designer, object designer, behavior designer, and representation implementer [HBR00].

Generator Developer. The generator developer is responsible to define the mapping from input model concepts to target language concepts [KT08] by choosing a suitable model-to-text transformation approach. He is also responsible to choose an appropriate generator architecture and implement the code generator. For example, in our scenario the goal is a modular architecture to support reuse.

In addition, the generator developer is also responsible to provide extension and adaptation points. Because of his knowledge about the code generator, he is also responsible to extend the code generator whenever required by the application developer.

A further concern of this role is the development of a RTE, which is required by the generated product to be fully functional.

Application Developer. The application developer is responsible for using the models provided by the modeler and the generator provided by the generator developer to develop a running application that fulfills the analyzed requirements by the modeler. This is achieved by generating the product and systematically and iteratively customizing the generated application. Similar roles have been identified in [MP03] (*application designer*) and in [Kul16] (*programmer*).

Since this role is only familiar with the generated product and its architecture, adaptations of the model and customizations of the code generator have to be handled by the modeler and the generator developer.

Besides customizing the generated product, the application developer is also responsible for realizing implementation concerns of the modeled business processes.

Senior Application Developer. The senior application developer is an advanced role of the application developer, who is able to adapt and extend the code generator but not forced to develop a new code generator. In addition, this role is familiar with the MLs used by the modeler and, hence, can adapt the models.

In general, a senior application developer is essential to reduce the learning curve of an MDE approach (cf. [BPRFF15]).

Infrastructure Provider. Each data-centric application requires a persistence infrastructure to persist the managed data. The persistence infrastructure is set up for each concrete product and is described in Section 7.5 in more detail. The essential tasks of this role involve setting up the infrastructure and performing data migration tasks, if required.

A similar role has been identified in [Wor16] (*run-time environment developer*) and [Kul16] (*administration role*).

In the proposed usage scenarios (cf. Section 3.1.1 and Section 3.1.2), no role for explicitly adapting the persistence infrastructure is considered, because it is assumed that the persistence is generic and does not contain any business logic but data persistence.

End User. The end user is familiar with the requirements and the required application. This role is concerned with receiving a product that helps to perform the desired tasks in his most comfortable way. A particular sub-role of the end user is the *customer*, who is not familiar with the needed application but in charge of financing the development. We do not explicitly consider this role, because the only concern of this role is the development time and costs.

As the proposed method is based on agile software development methods, the end user is constantly involved in the development process of the application for evaluation purposes.

3.2 Primary High-Level Requirements

From the envisioned usage scenario and the involved roles with their different needs, primary high-level requirements for a potential solution are derived in the following. They are grouped into four dimensions: general, modeling, generation, and generated

product requirements. Each of this high-level requirement is discussed and broken down into multiple fine grained requirements, general considerations, and architectural design decisions in the remainder of this thesis.

3.2.1 General Requirements

General requirements address considerations and issues that have to be regarded by the developed concepts, methods, and tools to realize the envisioned usage scenario. Subsequently, they are summarized.

RE-1 Effective MDD of data-centric applications. To improve efficiency of the data-centric application development, an effective MDD approach is targeted, i.e., generation of fully executable data-centric applications from CD4A models. However, not everything is generated but kept as a model-independent infrastructure, which "keeps the generator simpler" [KT08].

- **RE-1-1 Restricted type of target applications.** A necessity for effective MDD is the restriction of the type of target software system. Hence, it is restricted to data-centric applications with a three layered architecture on the client-tier (Section 2.3).
- **RE-1-2 Customization of generated source code.** To overcome limitations set by a restriction of the targeted software system (cf. RE-1-1) and, likewise, improve reusability, mechanisms to optimize and extend the generated source code are required [Sel03]. Hence, every generated source code artifact is individually customizable.

RE-2 Generic persistence infrastructure. To persistently store the managed data structure, a persistence infrastructure is provided. Rather than generating it from the input model, as proposed in [Loo17], a generic persistence infrastructure is used to enable rapid prototyping. It is set up by an infrastructure provider and can be reused in different generated products.

- **RE-2-1 Multi-tenancy and role-based access control.** The generic persistence infrastructure offers support multiple tenants and role-based access control [FKC07] for multiple collaborating users.
- **RE-2-2 Generic database schema.** To support different data structures, the persistence infrastructure has to provide a generic database schema.
- **RE-2-3 Consistent data migration.** Each generated data-centric application is considered as a single tenant storing its own data. To support migration of data between different generated versions of the same data-centric application, a consistent and model-specific data migration approach is provided.

RE-3 Support for iterative and incremental development: The MDD tools and methods to support the envisioned usage scenario explicitly facilitate iterative and incremental development. For tool support this implies that the customization, adaptation, and extension mechanisms have to ensure that manually-written source code is not manipulated during code generation. For development methods this means that generated code is not to be adapted by hand.

RE-4 Facilitate MDD tool reuse: MDD tools should be reusable for prototyping as well as for development of real-world data-centric applications. Hence, such tool reuse demands for inherent and high configurability, adaptability, and extensibility of the involved tools.

RE-5 Systematic and concise mapping. MDD's main assumption is that generated source code is never inspected [KT08]. Hence, the model is considered as the primary development artifact [BCW12]. Nevertheless, the generated source code has to be partially inspected, when customizations are necessary. Therefore, by providing a systematic and clear mapping from input models, the required knowledge about the generated source code is reduced to knowledge of the generated interface only.

RE-5-1 Provide mapping guidelines. Mapping guidelines are an essential element of mappings (cf. [HR00]), because whenever ambiguity appears, the chosen solution is always taken relative to the guidelines.

RE-5-2 Use defaults for semantic variation points in CD4A models. Although analysis models are considered to have a formal foundation because they are rooted on UML CD [SVC06], they contain semantic variation points [Grö10]. Such ambiguity demands for resolution to generate executable source code. Therefore, suitable defaults are provided that are made relative to the targeted data-centric applications (cf. RE-1-1) and mapping guidelines (cf. RE-5-1).

RE-6 Ensure data consistency. The data structure, which is implemented in Java, ensures consistency of its contained data. In general, data consistency (also considered as ensuring "semantic rules" [FBLS12]) refers to the property of the data structure, i.e., generated Java source code, to only allow objects that respect the defined CD4A constraints, i.e.,

1. the Java source code generated for a CD4A association ensures that the cardinality of mandatory associations is not violated, i.e., there is at least one association link for `[1..*]` and `[1]`; and at most one for `[1]` and `[0..1]`. (cf. ensuring multiplicity constraints [BFL13])
2. the Java variable implementing a CD4A attribute is assigned with a valid value, i.e., never `null`-values are assigned.

The formal definition is rooted on CD/OD consistency as described in [MRR11]. In more detail, the Java source code generated from a CD4A model *cd* ensures that every UML object diagram (UML OD) that can be instantiated from the generated Java source code is consistent to *cd*. To restrict semantic variability, we define that the empty UML OD is not considered and both the CD4A and the UML OD are complete. This understanding also assumes a well-formed CD4A model, which can be checked during language processing.

Due to missing CD4A language constructs to describe the quantity of qualifier keys (the cardinality of qualified associations defines the number of elements per key), it is assumed that the number of keys is not restricted. In consequence, the cardinality of a mandatory qualified association has to be ensured only if and only if keys exist.

Whenever data consistency is to be violated, we follow the fail-fast approach to prevent the data structure to become inconsistent, i.e., an exception is thrown and the data remains consistent. This design decision prevents instantiating objects of data structures that require temporary inconsistency, e.g., bidirectional association having a cardinality with a minimum of 1 on both ends. An approach to overcome this restriction is proposed in Section 5.3.

RE-7 Technical realization with MontiCore The developed DSL and tools are realized with the MC language workbench and code generation framework.

3.2.2 Modeling Requirements

Besides general requirements, which regard the overall usage scenario, modeling requirements for the developed MLs to describe the structural and behavioral aspects of data-centric applications are listed in the remainder of this section.

MR-1 Modeling language for analysis models: The envisioned usage scenario builds on analysis models for structural description of the managed data structure. To model such descriptions, a suitable ML that provides support for all required concerns but leaves out unnecessary ones is required (cf. [KKP+09]).

MR-2 Modeling language for business processes: To model processes on the managed data structure, an additional ML should be provided.

MR-3 Modeling language for an intermediate representation: To address the needs of generator developers to adapt code generators, this thesis proposes a transformation- and template-based code generation approach (cf. Chapter 8). This approach is based on an intermediate representation of the object-oriented structure of the generated source code. Hence, to represent the intermediate representation, a suitable ML is required.

3.2.3 Code Generator Requirements

When designing a code generator, the generator developer needs to decide on a model-to-text approach to generate concrete source code from the input model. Due to the different usage scenarios of the code generator, it has to facilitate the requirements set by modularity and customizability concerns. In the remainder of this section, the high-level requirements for such a code generator are summarized.

GR-1 Modular code generator architecture: The code generator is designed in a modular[1] way such that it is usable for prototyping, where the complete generated software system is used, and framework-like use, where only parts of the generated software system is used.

- **GR-1-1 Script-based configuration:** Modularity alone is useless, when there are no means to configure and use parts of the code generator. Hence, the code generator offers a script-based configuration approach to define its behavior including language processing and code generation.

GR-2 Code generator adaptation and extensibility: Design decisions (cf. RE-1-1) or defaults (cf. RE-5-2) of the code generator are not generally suited and, hence, demand for adaptation to address reuse (cf. RE-4). Therefore, the code generator uses an adaptable model-to-text transformation approach based on transformation- and template-based code generation.

- **GR-2-1 Extension via predefined hook points:** Extensions of the code generator can be realized via hook point-based adaptation, each of which is explicitly designed for predefined extension and is based on the idea of variation points [CN01, PBvdL05] and hook methods [Pre95].
- **GR-2-2 Extension via manually-written templates:** The code generator offers support to adapt and extend the code generation process by manually-written templates.

GR-3 Tracing capabilities: To improve code generator development, textual and graphical tracing information of the code generation process is provided in the form of generated reports.

GR-4 Execution of process models: Process models are executed by either generating source code or by interpreting it. To support the process modeling after deployment of a data-centric application, an integrated code generation and interpretation approach is developed that is based on previous research [Sar06, Ges10].

[1] Modularity is understood as the decomposition of the overall architecture into loosely coupled modules [RR15]. An understanding of the term module is given in Section 9.2.

3.2.4 Generated Product Requirements

Besides high-level requirements for the ML and the code generator, additional requirements for the generated applications are listed in the remainder of this section.

PR-1 Separation of concerns of generated source code: Separation of concerns is the approach to tackle complexity by decomposing it [Dij82]. Hence, the generated source code is structured according to its purpose (cf. RE-1-1). This approach also promotes modular use of the generated source code (cf. GR-1).

PR-2 Graphical management of data: The generated data-centric application provides a fully functional GUI to manage the data. Furthermore, it supports user login, user management, and execution of processes.

- **PR-2-1 Support for undo/redo.** The generated GUI supports undo and redo functionality by default using the model-specific Command-Pattern [GHJV95].
- **PR-2-2 GUI thread management.** To support responsiveness of the GUI, the generated application supports GUI threads that manage communication with the persistence infrastructure.
- **PR-2-3 GUI actions cannot violate data.** The GUI supports management functionality for the data structure but does not allow to store invalid data.

PR-3 Scalable object management: A management facility that manages all objects in a centralized way has to address scalablility, which demands for approaches to lazily request chunks of data from the persistence infrastructure. At the same, this additional complexity needs to be hidden from application developers.

PR-4 Predefined customization via hot spots: The generated software system provides a set of hot spots, which represent predefined spots in the generated software system intended for customization [Pre95].

PR-5 Persistent and non-persistent mode: While a data-centric application has to provide means for persistent storage (cf. RE-2), data-centric application prototype does not require such support, because only the functionality is demonstrated. As a consequence, the data-centric infrastructure has to provide a mode for persistent storage and one mode for a non-persistence storage.

3.3 Envisioned Methods for MDP and MDD of Data-Centric Applications

The envisioned development and prototyping methods are realized by the MontiDEx code generator and the MontiDEx product, as shown in Figure 3.5. In the remainder of this section, an overview of the developed tooling and methods is given.

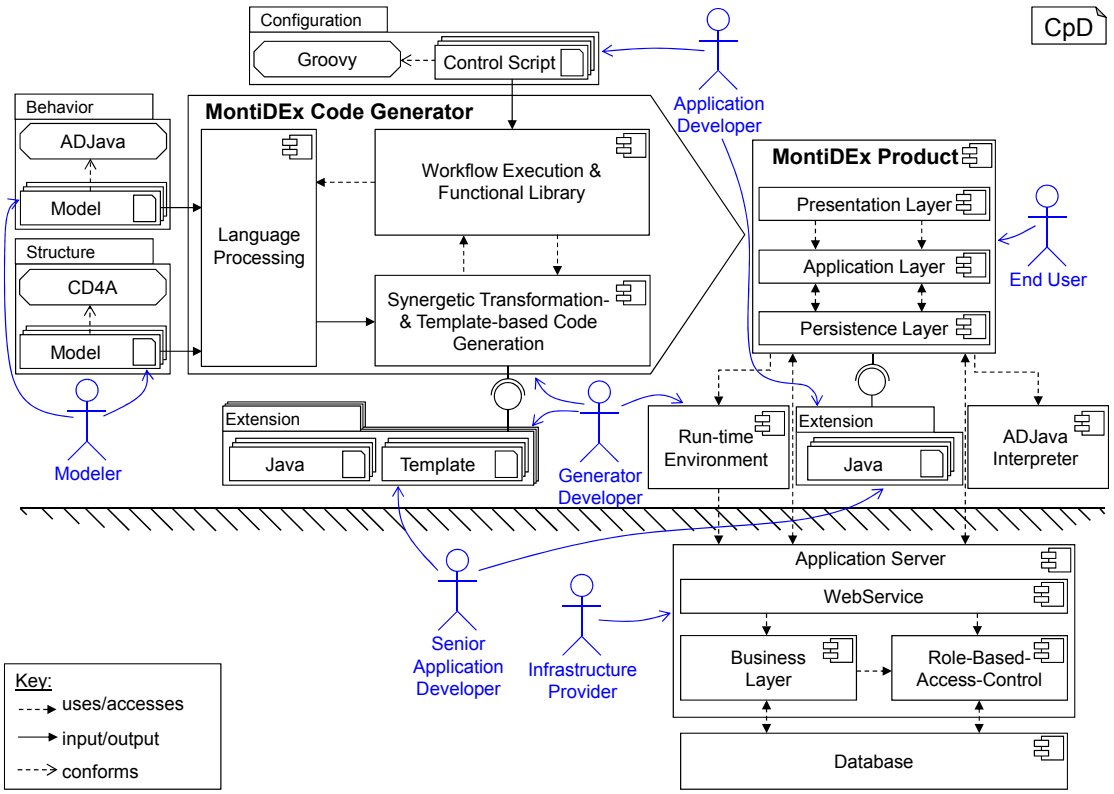


Figure 3.5: Overview of the MontiDEx code generator and the MontiDEx product.

The input of the MontiDEx code generator are CD4A models, ADJava models, and Groovy scripts [KGK+07]. Each CD4A describes the structural properties of the managed data. An ADJava model describes the processes of data-centric applications and is an instance of a DSL for UML ADs with embedded Java. Each CD4A and ADJava model is processed by the `Language Processing` component, which is responsible to create the corresponding ASTs. Finally, the Groovy configuration script defines the pipeline of language processing and code generation steps by using workflows and predefined functionality defined in the `Workflow Execution & Functional Library` component, which is explained in Section 2.2

The code generation in the MontiDEx code generator is realized in the `Synergetic Transformation- and Template-based Code Generation` component. It uses an integrated approach of transformation- and template-based code generation as explained in Chapter 8. This model-to-text transformation approach supports black-box extensions and adaptations of the overall code generation process by facilitating manually-written transformations and templates.

The MontiDEx code generator generates a fully executable data-centric application from the input models. Each generated MontiDEx product is realized as a three-layered architecture providing a GUI to manage the modeled data structure (cf. Chapter 7). To persistently store the created objects, a generic persistence infrastructure is used (cf. Section 7.4). This infrastructure (`Application Server` component) allows to efficiently create multiple instances and also supports multiple roles and rights for users. Furthermore, it is able to manage multiple tenants. In addition, the generate application uses a RTE (`Run-time Environment` component), which contains non-generated code to make the generated code functional, and an interpreter for the ADJava ML (`ADJava Interpreter` component) to support process automation.

3.3.1 MDP of Data Structures with MontiDEx

A method to use the developed concepts and tools to support MDP is shown in Figure 3.6. It is based on already established approaches [MP03, BDLD11] and illustrates the essential activities and roles. This method is a lightweight approach and, hence, may be extend with additional activities in requirements engineering, e.g., [Som10]. Furthermore, it is assumed that the domain is unknown, no CD4A model or prototype implementation is present, and GUI prototyping is not addressed in contrast to, e.g., [SS16, BPRFF15]. Moreover, we assume that application developers implement and test added functionality and, hence, omit an additional testing activity. As a result, possible limitations may occur if the involved personnel is not skilled or motivated (cf. [MP03]).

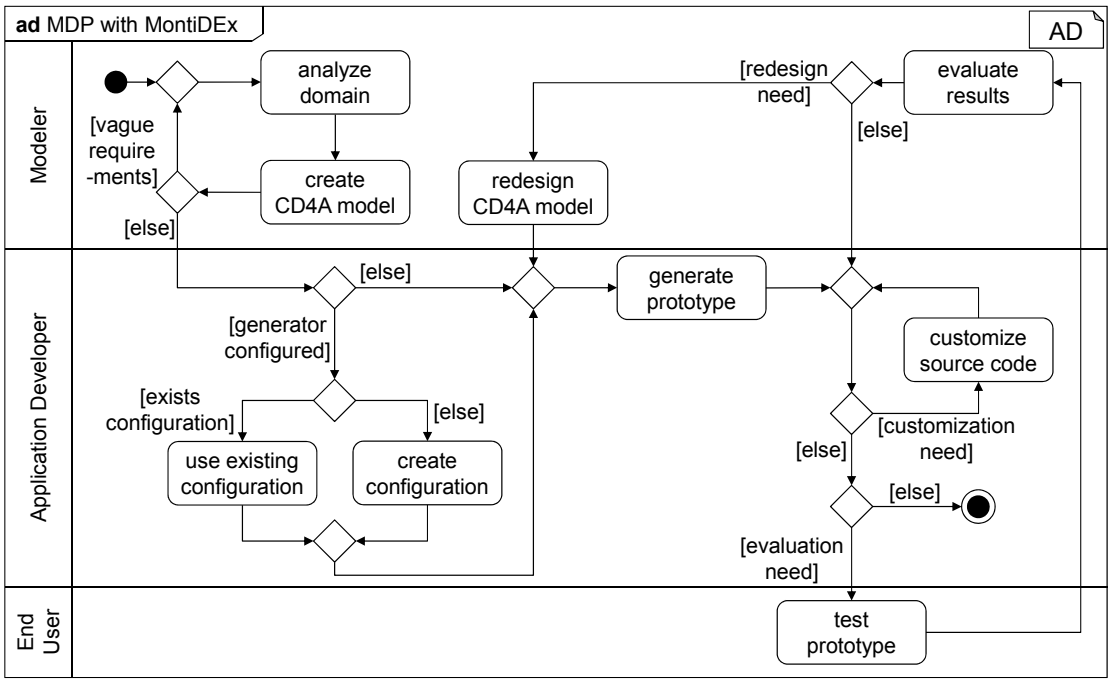


Figure 3.6: A Method to use the developed concepts and tools for MDP.

Each prototype development starts with an analysis of the domain and the requirements (*Agile Analysis* phase in [BPRFF15]), which is performed by the modeler. The goal of this activity is to understand the software system under development and retrieve the required information to allow the modeler to create a structural model of the managed data structure using the CD4A ML. Vagueness during CD4A model creation can be resolved via underspecification in the model or by questioning the modeler. In this step, it may be required to decompose the overall systems into smaller sub-systems as proposed in [MP03], each of which can be prototyped separately by applying the subsequent activities.

Analysis and modeling is required to generate a prototype from the CD4A model. For this purpose, the MontiDEx code generator has to be set up by either using one of the provided code generator configurations or create a custom configuration as described in Section 9.3. Afterwards, the application developer generates a prototype, i.e., MontiDEx product, by executing the MontiDEx code generator. If needed the functionality of the MontiDEx product can be customized using the predefined approaches to add manually-written code, as described in Chapter 6. For prototyping, it is sufficient to adapt generated source code only, because the prototype is intended to be thrown away and reuse is not addressed. Therefore, extension and adaptation of the code generator are not required during prototyping.

An essential element of this proposed method is that customizations are added in separate artifacts using the target language, as explained in Chapter 6. As a result the generated source code is regarded as a disposable product (cf. [SVC06]), because it can always be reproduced by the MontiDEx code generator and the CD4A model.

The synthesized and customized prototype is tested by the end user. The focus of this activity is on elucidating requirements of the data structure. All results gained during prototype testing are evaluated by the modeler and may yield a redesign of the CD4A model and additional customizations of the generated source code. However, a redesign of the CD4A model may influence manually-written customizations, e.g., deletion or renaming of CD4A classes. Hence, already manually-written source code may have to be adapted to the redesign.

Eventually, this prototyping process is ended if further prototype evaluation is not needed and the data structure is identified. The resulting CD4A model developed during MDP can be used for MDD of data-centric applications as it is described in the subsequent section.

3.3.2 MDD of Data-Centric Applications with MontiDEx

A method to support agile MDD of data-centric applications using the developed concepts and tools is illustrated in Figure 3.7. It is assumed that the domain, the requirements, as well as the structured information to be managed are known. Both can be identified using the prototyping approach presented in Section 3.3.1.

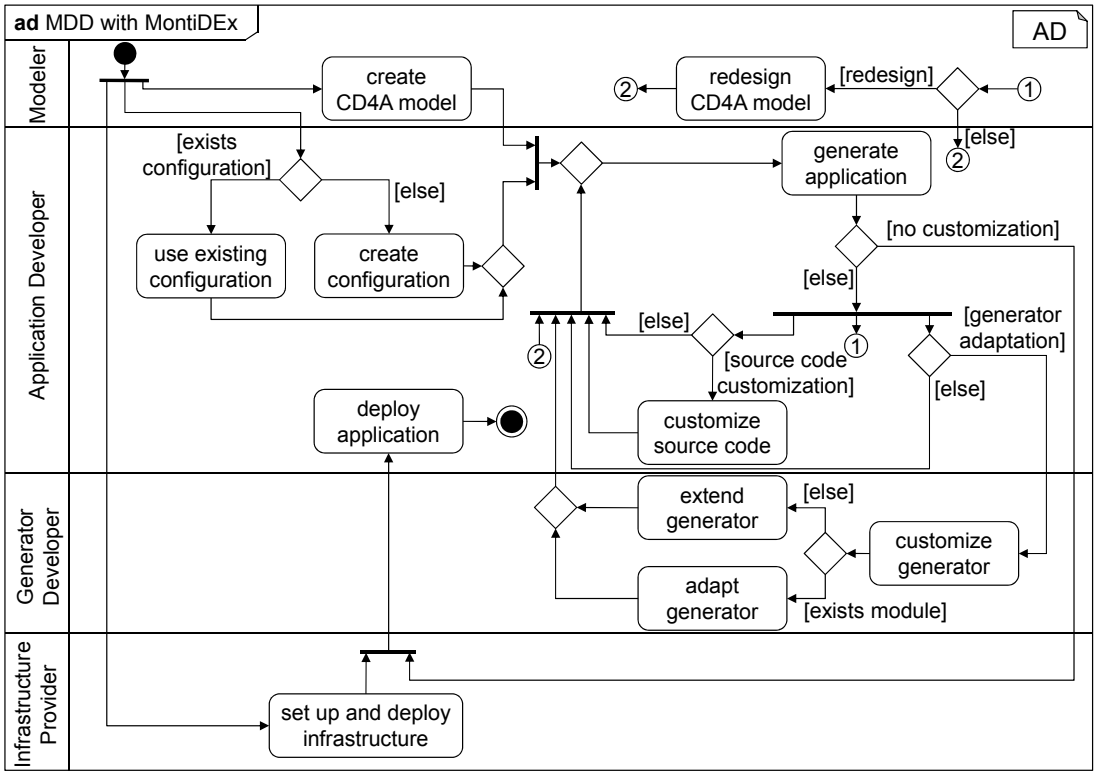


Figure 3.7: Method to use the MontiDEx generator and the MontiDEx product for MDD.

The development method is started by the modeler creating a CD4A model. As the basis for creating a model, the model for prototyping (cf. Section 3.3.1) can be used. At the same time, the code generator has to be set up by the application developer by creating either a new configuration or designing a custom one. In addition, the infrastructure provider has to set up the persistence infrastructure for the development team and also a persistence infrastructure for productive use, which is deployed to the end user. In contrast, the method for MDP of data-centric application prototypes does not require such an infrastructure, because each MontiDEx product provides a non-persistent mode that is fully functional but only used for demonstration purposes Section 7.5.

In addition to creating a CD4A model, it may be required to develop a model for business processes. Such models can be created by a modeler using the provided AD-Java ML presented in Chapter 10. A method to extend data-centric applications with processes is illustrated in Section 10.4.

The actual development of the software system starts after the first code generation has been performed by the application developer. Since the generated application only provides basic management functionality, it has to be customized. This includes cus-

tomizations of the generated source code as well as customizations of the code generator itself. However, while source code can directly be adapted by the application developer, only the generator developer or the senior application developer role can adapt or extend the code generator. Code generator adaptation is required when defaults or architectural design decisions have to be adapted; manually-written customization of the source code is not practical; or reuse of the provided functionality is intended. Such adaptations and extensions include adapting the code generation process, as described in Section 8.2.4, as well as adapting the configuration script, as shown in Section 9.3. It is essential to notice, that during the cycle to add functionality, the CD4A model can always be redesigned.

The customizations, adaptations, and extensions may have to be performed multiple times to meet the end user's requirements. This cycle ends when all requirements are fulfilled. Then, the application is deployed by the application developer. Deployment involves packaging on of the different MontiDEx projects, as described in Section 9.5.2, and deploying the infrastructure to the end user's target environment.

Chapter 4

UML Class Diagrams in Analysis, Design and Implementation

MDP and MDD of data-centric applications, as proposed in Chapter 3, is based on a notion to describe the structure and interconnections of the managed data. The UML/P CD ML provides a suitable notion for creation of descriptive and prescriptive structural models [Sch12, Rum16]. It also removes semantically underspecified language concepts to enable the use of such models as a primary development artifact. However, due to UML/P CD's focus to support all UML CD language concepts, it is not suitable for specifying analysis models. Hence, the language concepts have to be restricted. This is necessary, because unnecessary generality hampers its successful use of DSLs (cf. [KKP+09]). Likewise, the same holds for DSLs supporting too few language concepts (cf. [KKP+09]). Hence, in this thesis the CD4A language, which is based on UML/P CD and suits the needs of analysis and domain models has been developed (cf. *MR-1*). In addition, to support implementation models, which are UML CD models using all language concepts and are essential for the targeted code generation approach (cf. *MR-3*), the CD4Code ML is has been developed as an extension of the CD4A ML.

This chapter aims to introduce the CD4A ML to modelers (cf. Section 3.1.3), whereas the description of the CD4Code ML primarily addresses generator developers (cf. Section 3.1.3). Hence, in this chapter, a clear understanding and definition of analysis, design, and implementation models is presented and the supported language concepts of analysis models are explained in Section 4.1. Afterwards, the CD4A language for analysis models is introduced in Section 4.2. Finally, the CD4Code ML, which is an extension of the CD4A ML, is presented in Section 4.3.

4.1 Analysis, Design, and Implementation Model

Models in software engineering can be divided into *analysis models* and *design models* [RBP+91], because of the common separation of the software life-cycle into an *analysis* and a *design phase* [HS93]. While analysis models describe an understanding of a problem, design models target creation of a solution for the analyzed problem [GVM09]. For example, UML CD in the analysis phase are used to structure concepts of the real

world, whereas UML CD in the design phase are used to represent the structural view of a system [Rum16]. Hence, in the analysis phase "models are usually general and with no information that can lead to a technical solution" [BPRFF15].

This separation of models with respect to their purpose is applied in model-driven architecture (MDA), which is an Object Management Group (OMG) standardization of MDD [BCW12]. In this context, an analysis model can be regarded as a PIM, whereas a design model represents a PSM (cf. [Mel04]).

Following this understanding and separation of models in software engineering, an analysis model is defined in this thesis as:

Definition 11 (Analysis Model)**.** *An analysis model is a model that describes an understanding of the problem in a domain using a descriptive and highly abstract description.*

This definition of an analysis model is too broad such that a variety of models suit this understanding. A similar definition has already been proposed in [KGBE06] with the same effect. In consequence, in this thesis the set of analysis models is restricted to UML CD models, because the aim of this thesis is a structural description of managed data. Furthermore, such UML CDs have to be regarded as analysis models by the proposed classification approach in [GVM09], which proposes three orthogonal dimensions (*reality, purpose, and abstraction*) for the separation of analysis and design models. Note that for such analysis models the term *conceptual class model* or *domain model* is also used. In this thesis, these terms are considered as equal. However, to separate prototyping and development, we use the term analysis models for prototyping and domain model for development of data-centric applications.

While classes in analysis models represent real world concepts, classes in design models describe code fragments [HS93, Kai99, Fow03b]. This characteristic of a design model is manifested in the following definition, which is used throughout this thesis:

Definition 12 (Design Model)**.** *A design model is a model specifying a solution (usually a software system) for the analyzed problem.*

Just as for analysis models, the set of design models is reduced to UML CDs that are regarded as design models according to the understanding in [GVM09]. Note that although an analysis and a design model describe two different realities, it is possible to transform an analysis model into a design model (cf. [GVM09]).

A design model is by definition not complete in terms of describing every aspect of the implementation. It only represents a part of it. For example, it does not necessary have to contain accessor and mutator methods. Hence, we define the *implementation model* as follows:

Definition 13 (Implementation Model)**.** *An implementation model is a design model that fully represents a software system's detailed (usually object-oriented) structure.*

Since an implementation model is a design model, implementation models are also regarded as UML CD models in this thesis. Therefore, implementation models describe only the object-oriented structure of the technical realization but omit implementation details such as method bodies. In general, implementation models support the same language concepts as design models, which are a superset of the ones supported by analysis models.

4.1.1 Language Concepts in Analysis Models

While design models require all UML CD language concepts (cf. [Sco04]), because they serve a particularizing purpose to describe a concrete system in a specifying way, analysis models only require a subset of the available UML CD language concepts to describe a data structure. Hence, in the remainder of this section, the required UML CD language concepts that have to be supported by a DSL for analysis models are presented. Note that for presentational reasons a detailed description of each UML CD language concept is neglected. Instead, interested readers are advised to refer to [Rum12, Sch12, Rum16].

To structure concepts of the real world, analysis models use classes, interfaces, and enumerations (cf. [RBP+91, EKW92, MS92, Sco04, Jac04]). While classes represent real or abstract concepts of the problem domain, the main purpose of interfaces in the analysis phase is to represent *markers* denoting properties or structure concepts (cf. [Fow97]). Enumerations represent a group of values. As a result, inheritance makes semantically sense for classes and interfaces only (cf. [WN94]).

Based on this use of interfaces, classes, and enumerations, only classes comprise attributes. An attribute's type can be primitive [RBP+91, CAB+94, Jac04], or a complex type such as other classes, interfaces, or enumerations. Moreover, attributes can also be derived to denote that the attribute's value can be calculated ("derived") from other attributes, objects, or associations [Rum16]. Likewise, values can be assigned to attributes to denote initial values.

To represent relationships between interfaces, classes, and enumerations, analysis models support associations and composition [CY91]. In addition, each relationship may have a particular navigation direction.

For all UML CD language concepts in an analysis model, visibility is not needed, because in an analysis model nothing is hidden (cf. [EKW92]). Likewise, modifiers such as *final* and *static* are not required, because they regard solution concerns.

4.2 CD4A: Modeling Language for Analysis Models

The *UML class diagram for analysis (CD4A)* ML has been developed in this thesis to address the requirements of analysis models (cf. Section 4.1.1). Conceptually, it can be seen as a language extension of the UML/P CD language by means of restrictive context conditions (cf. Section 2.2.1). However, technically it is realized as a new DSL using the

current version of MontiCore (cf. Section 2.2), which is not compatible with the version used in UML/P CD. The full grammar is shown in Listing C.1. Note that an overview of alternative existing DSLs for UML CDs is given in [SG16].

To explain the CD4A language, the UML CD shown in Figure 4.1 is used. It describes a simplified banking system. In this example, the modeled banking system contains different customers (`Customer`), their transactions (`Transaction`), and different kinds of accounts (`CheckingAccount` and `SavingsAccount`). A bank may additionally provide deposits (`Deposit`) to allow customers to trade with shares (`Share`). Note that monetary amount is modeled as an integer in cent to avoid rounding differences.

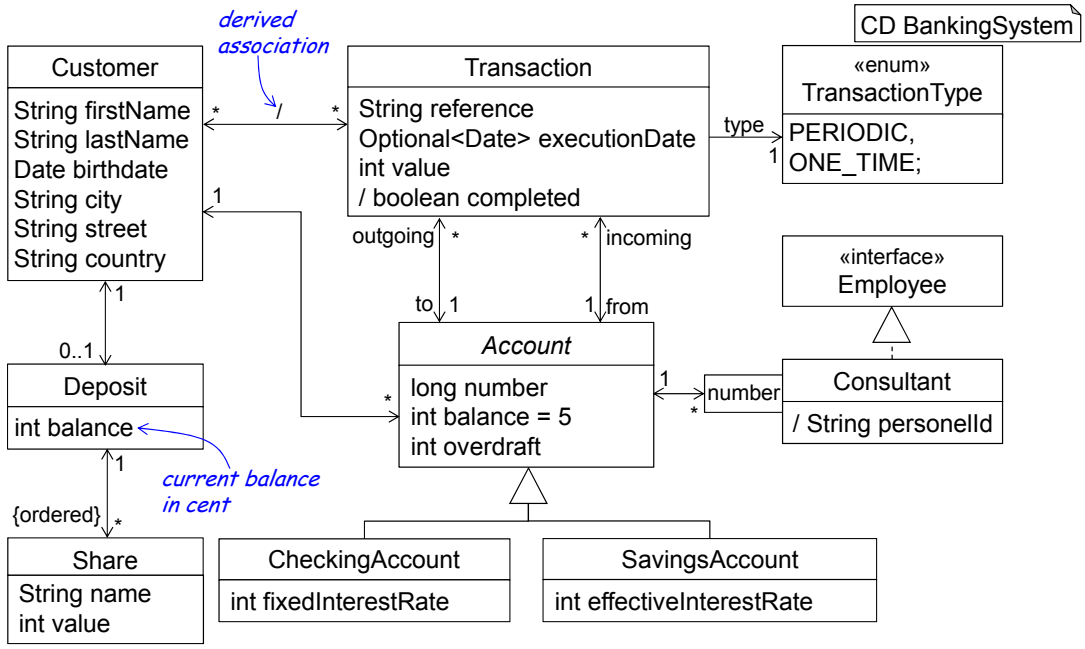


Figure 4.1: A UML CD for a simplified banking system.

Subsequently, the CD4A model for the example in Figure 4.1 is gradually created and explained. For presentational purpose, the relevant parts of the full CD4A model are shown as excerpts marked by the model's name. To explain further language concepts, which are not part of the example, the model name is omitted. The complete listing of the final CD4A model is shown in Listing D.1.

4.2.1 Model Definition

A CD4A model is defined in one single file, which has to have the same name as the model. This design decision corresponds to proposed approach in [Sch12] to improve the tracing of models to its containing artifacts.

Each CD4A model may start with a **package** keyword followed by a qualified name to define the package of the model but not necessarily of the contained classes, as shown in the example in l.1 in Listing 4.1. It helps to structure multiple models into logical units and group all later used names in the artifact to avoid conflicts (cf. [Sch12]).

Besides the package, external artifacts that contain data types, i.e., types not defined in the model (subsequently called *external data types*), can be imported using the **import** keyword followed by a qualified name, e.g., Java classes such as `java.util.Date` in l.3. Each external data type has to be imported to respect referential integrity (cf. [Sch12]).

```
1 package dex;                                          CD4A BankingSystem
2                                                                ...
3 import java.util.Date;
4
5 classdiagram BankingSystem {
6   //...
7 }
```

Listing 4.1: A model definition for the banking system example in Figure 4.1.

The main part of a CD4A model is introduced by the **classdiagram** keyword followed by the diagram's name, e.g., in Listing 4.1 the model named `BankingSystem` is defined in l.5. The subsequent brackets enclose interfaces, classes, enumerations, and associations that belong to this model.

4.2.2 Interfaces, Classes, and Enumerations

An interface is defined by the **interface** keyword followed by a name, as shown in Listing 4.2. Interfaces in the CD4A language have no modifiers such as public or protected, no attributes, and no methods. The reason for this design decision is rooted in the purpose of CD4A to represent analysis models (cf. Section 4.1).

```
1 interface Employee;                                   CD4A BankingSystem
2                                                                ...
```

Listing 4.2: A CD4A model showing the definition of an `Employee` interface.

Each interface may extend one or multiple other interfaces using the **extends** keyword, as shown in Listing 4.3.

```
1 interface A;                                                    CD4A
2 interface B extends A;
3 interface C;
4 interface D extends B, C;
```

Listing 4.3: CD4A supports interface hierarchies using the **extend** keyword.

Besides interfaces, CD4A allows the definition of multiple classes. Each class is defined by the **class** keyword and a name, as depicted in l.1, l.3, and l.5 in Listing 4.4. An optional **abstract** keyword preceding the class definition denotes that the class is abstract, as shown in l.1. In addition, each class can extend one other class, which is denoted by the **extends** keyword (l.3), or can implement multiple interfaces listed after the **implements** keyword (l.5). Note that multi-inheritance is explicitly forbidden.

Just like interfaces, classes in CD4A do not support visibility. A CD4A class does also not allow for any other modifier than **abstract**. In addition, methods and constructors are explicitly forbidden.

```
1 abstract class Account { ... }                    CD4A BankingSystem
2                                                          ...
3 class CheckingAccount extends Account { ... }
4
5 class Consultant implements Employee { ... }
```

Listing 4.4: CD4A classes can be abstract (l.1), can extend other classes (l.3), or implement interface (l.5).

In addition to interfaces and classes, an enumeration can be defined with the **enum** keyword and a name (l.1 in Listing 4.5). Each enumeration groups a set of values. Hence, CD4A enumerations cannot have any constructor, method, or even an attribute. Moreover, they cannot inherit from interfaces, classes, or enumerations and cannot have any modifier (cf. Section 4.1.1).

```
1 enum TransactionType {                            CD4A BankingSystem
2   PERIODIC, ONE_TIME;                                    ...
3 }
```

Listing 4.5: A CD4A definition of the `TransactionType` enumeration.

4.2.3 Attributes and Predefined Data Types

Each CD4A class may bundle a set of attributes, each of which has a type and a name as shown in Listing 4.6 in ll.2-4 and l.8. Derived attributes are defined using a leading **derived** keyword or the **/**-symbol, as shown in l.8. Note that this is the only modifier supported. In addition, an attribute can be initialized with a default value of its type, e.g., l.3. Only primitive and String values are allowed but no expressions. Semantically, this means that the value is assigned when creating an object but can later be changed.

```
1 abstract class Account {                                  CD4A BankingSystem
2   long number;                                                ...
3   int balance = 5;
4   int overdraft;
5 }
6
7 class Consultant implements Employee {
8   / String personelId;
9 }
```

Listing 4.6: A CD4A definition of attributes within classes.

The attribute's type can be an arbitrary type defined in the model or an imported external data type. In addition, type constructors such as `List<.>`, `Set<.>`, `Optional<.>`, and the primitive Java data types as well as their wrapper types are predefined, i.e., do not have to be imported explicitly. For example, l.7 in Listing 4.7 shows the use of the `A` class as a generic type parameter for `Optional`. Besides wrapper and complex types for generic collection types and optional data types, CD4A allows to use an arbitrary primitive data type as a generic type parameter, e.g., `List<int>` in l.2.

```
1 class A {                                                 CD4A
2   List<int> intValues;
3   Set<String> stringValues;
4 }
5
6 class B {
7   Optional<A> optValue;
8 }
```

Listing 4.7: CD4A supports the predefined data types `List<.>`, `Set<.>`, and `Optional<.>`.

4.2.4 Associations

Relationships between interfaces, classes, and enumerations are represented by associations, each of which is defined by the **`association`** keyword, a left association end, a navigation direction, and a right association end. Due to the focus of this thesis and the requirements set by analysis models, associations are limited to binary associations with navigability, ordering, and qualification.

Each association can have a cardinality defined on both association ends, as shown in Listing 4.8. CD4A supports four different types of cardinalities: `[1]`, `[0..1]`, `[*]`, and `[1..*]`, which are the most commonly used cardinalities in practice (cf. [BFL13]). If further restrictions on the cardinality are needed, they can either be realized in the

design phase or by using the UML/P object constraint language (UML/P OCL) constraints [Rum12].

```
1 association [1] Customer     <->   Deposit [0..1]; CD4A BankingSystem
2 association [1] Account (to) <->                          ...
3                         (outgoing) Transaction [*];
4 association type Transaction  ->   TransactionType [1];
```

Listing 4.8: An association is defined by an **association** keyword, a cardinality, a navigation direction, an association name (l.4) and role names (ll.2-3).

In general, cardinalities are optional and, hence, can be omitted. Semantically, this represents an underspecification, which has to be handled by a default understanding. Note that cardinalities on non-navigable association ends (left association end in l.4) can be omitted, as they are hard to realize.

Each association can have an association name and a role name on both ends. The primer is defined after the **association** keyword as shown in l.4 in Listing 4.8, whereas role names are defined in round brackets on the left or the right hand side of the navigation direction, as shown in ll.2-3. Explicit role names and association names are necessary whenever ambiguity regarding the derivation of the association's name occurs. For example, when dealing with reflexive associations. Association names can be omitted, because they can be derived by using the opposite reference name.

Besides, both association ends have a particular reference type, which can be an arbitrary model type, i.e., an interface, a class, or an enumeration, defined in the current CD4A model or in another model but imported via the **import** statement. For example, all associations in Listing 4.8 show reference types defined in the current model only. In addition, it is possible to use external data types as reference types, as shown in Listing 4.9. However, primitive data types are not supported but only their wrapper types, e.g. l.2. Likewise, type expressions using collection types or generic types such as `Optional<.>` cannot be used in a reference type of association ends.

```
1 association birthday A -> Date;                    CD4A
2 association name     A -> Integer [1];
```

Listing 4.9: Associations to external data type `Date` (l.1) and `Integer` (l.2).

To denote the navigation direction, which describes that only properties of the element in the direction of the navigation can be accessed (cf. [Gén01, Rum12, Rum16]), one of the following navigation directions can be used: <->, <-, ->, or --. Each symbol represents the navigation direction in the direction of the formed arrow. The latter navigation direction (--) represents an underspecification.

Navigation directions are restricted when used for enumerations and external data types such that it is not possible to navigate from an external data type, as subsequently described in more detail in Section 4.2.5.

Derived Association

As well as attributes, associations can be derived. Semantically, they are used to represent that the set of association links can be calculated from other elements [Rum16].

Derived associations are denoted with a derived modifier (**/**-symbol or **derived** keyword) after the **association** keyword, e.g., the association in Listing 4.10.

```
1 association / [*] Transaction <->                    CD4A BankingSystem
2                   Customer [*];                              ...
```

Listing 4.10: Derived associations are denoted with a **/**-symbol.

Note that the derived modifier does not depend on the association's type or navigation direction as shown in Listing 4.11. Yet, it is the only supported modifier.

```
1 association / name      A  -> B [1];                          CD4A
2 association derived [1] A <-> B [0..1];
```

Listing 4.11: Every CD4A association can be marked as derived by a **/**-symbol (l.1) or a **dervied** keyword (l.2).

Ordered Association

In general, stereotypes are used to specialize model elements (cf. [Rum16]) but the concrete semantics of a stereotype can be defined during implementation. Hence, stereotypes in analysis models do not represent technical concerns.

CD4A supports the «`ordered`»-stereotype that can be attached to associations ends with cardinality `[*]` or `[1..*]` to preserve the order in which items are added. For example, the association shown in Listing 4.12 is ordered in the direction from `Deposit` to `Share`.

```
1 association [1] Deposit <->                          CD4A BankingSystem
2                 Share [*] <<ordered>>;                       ...
```

Listing 4.12: Associations can be marked to preserve the order of association links with the «`ordered`»-stereotype.

This stereotype can be attached to each association end individually. Yet, semantically it makes sense for `[*]` or `[1..*]` cardinalities only. For associations with cardinality `[1]` or `[0..1]`, this stereotype is ignored.

Qualified Association

Another type of associations are qualified associations, which define the access to an instance from a set of instances by using a particular qualifier, which has been previously assigned to this instance (cf. [Rum12, Rum16]). In other words, each association link is added as a tuple consisting of a qualifier and the association link. Since qualified access is independent of technical considerations, it has to be mapped to equivalent representations in the implementation phase.

A qualifier is either an attribute's value of the opposite instance or an arbitrary value of a certain type. For example, Listing 4.13 shows a qualified association with the `[[number]]` qualifier. The two square brackets denote that this qualifier is an attribute of the reference in the navigation direction, i.e., in this particular example, `number` has to be an attribute of the `Account` class or its super class.

CD4A BankingSystem

```
1 association [1] Account <->
2                 [[number]] Consultant;
```

Listing 4.13: A qualified associations using an attribute value as the qualifier.

An arbitrary value of a certain type for a qualifier is denoted with only one bracket, e.g., `[String]` in l.2 and l.3 in Listing 4.14. For this type of qualifier any arbitrary value of the qualifier's type is valid, i.e., in this case any `String` value. Note that the qualifier is supported on both association ends, as shown in l.3. Furthermore, as qualified associations are a special kind of association, they may have cardinalities on both ends and may even be marked as ordered, e.g., l.2, or derived (l.3).

CD4A

```
1 association   A [[number]] <-> B [1];
2 association   C [String]    -> D [*] <<ordered>>;
3 association / E            <-> [String] F;
```

Listing 4.14: An example of supported qualified associations.

Each cardinality refers to the number of association links referenced by the qualifier. CD4A does not provide means to quantify the amount of qualifiers. As a consequence, this denotes an underspecification in CD4A. Moreover, bidirectional qualified associations on both association ends are not allowed, because of the chosen mapping in this thesis (cf. Section 5.2.9).

Composition

A special kind of an association is the composition. In a composition the composed instances are in a strong relation that also involves the life-cycle [Rum12, Rum16]. A composition is defined similar to an association but with the first keyword being **composition** rather than **association**. The left composition end forms the containing class (whole) end right composition end forms the contained class (part). The cardinality of a composition from the part to the whole is restricted to `[1]`.

An example of a composition is shown in Listing 4.15. In l.1, `A` class is the containing class and `T` is the contained class. This example also shows that compositions can be derived (l.2) and have qualifiers (l.3).

```
1 composition ordinary  A             -> T [0..1];
2 composition / [1]     A            <-> T [1];
3 composition qualified A [[attr]]   <-> T [1];
```

CD4A

Listing 4.15: The **composition** keyword denotes a composition that can be derived (l.2) or qualified (l.3).

4.2.5 Context Conditions

The aforementioned CD4A language concepts are intended to demonstrate the concrete syntax of the language. However, these language concepts allow to create semantically invalid models, e.g., use of an attribute type that is not imported or defined. Therefore, in the remainder of this section a set of context conditions (cf. Section 2.2) is presented that restricts the set of CD4A models to semantically valid ones.

Due to presentational reasons, the following description is limited to the main context conditions with their error code only. The complete list is shown in Section E.1.

Diagram Name

The set of diagram names is restricted to those that do not start with a lower case or a numerical character (Error Code: 0xC4A01). Even if this context condition is purely based on common conventions (cf. [Sch12]), it is regarded as an error (Severity: error). The example in Listing 4.16 shows a CD4A model with an invalid name, because it starts with a lower case.

```
1 classdiagram bankingSystem{ }  //0xC4A01
```

CD4A

Listing 4.16: An example of a CD4A model with an invalid diagram name.

Another context condition, related to the diagram's name, requires that each CD4A model is located in one particular artifact (Error Code: 0xC4A02), as already mentioned in Section 4.2.1. This convention is rooted on the Java programming language, where the compiler enforces one type per file due to possible file system conflicts (cf. [Sch12]). In this context, its main purpose is to improve traceability, i.e., prevention of misunderstandings when reading and editing models. Hence, the severity of this context condition is a warning (Severity: warning), because the model can still be processed but further processing including symbol table creation may result in an error.

Interfaces, Classes, and Enumerations

Equally for all interfaces, classes, and enumerations, their name has to be unique through the whole model as they represent an identifier for a particular type (Error Code: 0xC4A04). Consequently, to avoid unambiguity this context condition is considered as an error (Severity: error), if non-unique names are detected, as shown in ll.2-3 in Listing 4.17. Moreover, in compliance to the context condition regarding the diagram's name, the name used for an interface, a class, or an enumeration has to start with an upper case (Error Code: 0xC4A05). Violating this context condition is regarded as an error (Severity: error). For instance, as shown in ll.4-5 in Listing 4.17.

```
1 classdiagram NonUniqueName {                                    CD4A
2   interface Person;
3   class Person;  ✗              //0xC4A04
4   interface employee;  ✗        //0xC4A05
5   abstract class account;  ✗    //0xC4A05
6 }
```

Listing 4.17: An example of an invalid CD4A model because of non-unique and lower case type names.

Another type of restriction regards enumeration constants. Each enumeration constant has to be unique (Error Code: 0xC4A06) and the name has to be written in capitalized characters (Error Code: 0xC4A90). As shown l.5 and l.6 respectively in Listing 4.18, violations of these restrictions are handled as errors (Severity: error).

```
1 classdiagram NoUniqueEnumConstant {                             CD4A
2   enum T {
3     PERIODIC,
4     ONE_TIME,
5     PERIODIC, ✗  //0xC4A06
6     regular; ✗   //0xC4A90
7   }
8 }
```

Listing 4.18: A CD4A model with an invalid definition of enumeration constants.

Extends

The **extends** language concept, which denotes that a model type extends another model type by inheriting properties of its parent (cf. [Rum12, Sch12, Rum16]), allows to extend arbitrary types. However, semantically this does not always make sense, as described in Section 4.1.1. Hence, we restrict the set of valid CD4A models by explicitly forbidding (Severity: error) circular **extends**-relations (Error Code: 0xC4A07). An example of an invalid CD4A model is depicted in Listing 4.19.

CD4A

```
1 classdiagram CircluarExtends {
2   abstract class A extends C;
3   class C extends A; ✗ //0xC4A07
4 }
```

Listing 4.19: An example of a CD4A model with a circular inheritance.

Another restriction is that classes can only extend other classes but no enumerations and interfaces (Error Code: 0xC4A08), e.g., l.4 and l.9 in Listing 4.20. This kind of modeling errors (Severity: error) complies to the understanding in object-oriented programming languages including Java and the understanding in UML/P CD. Moreover, enumerations are not allowed (Severity: error) to extend or implement any type defined in the model as well as any external data type (Error Code: 0xC4A10), e.g., l.5. An invalid model showing these errors is depicted in Listing 4.20.

CD4A

```
1 import java.util.Date;
2 classdiagram InvalidExtends {
3   interface E;
4   class C extends E; ✗          //0xC4A08
5   enum T extends E { ✗          //0xC4A10
6     PERIODIC,
7     ONE_TIME;
8   }
9   class A extends T; ✗          //0xC4A08
10  class MyDate extends Date; ✗  //0xC4A92
11  interface B extends MyDate; ✗ //0xC4A09
12 }
```

Listing 4.20: An example of a CD4A model with an invalid `extends`-relation.

The CD4A grammar (cf. Listing C.1) allows a superclass to be of any type including external data types such as `java.util.Data`. This may, however, require processing the external data type when used for code generation to generate syntactical correct source code. Hence, to avoid such errors, CD4A forbids (Severity: error) external types as

superclasses (Error Code: 0xC4A92) as shown in Listing 4.20 in l.10. These restrictions can be overcome by using the method presented in Section 6.4.1.

For interfaces, the **extends**-relation defines that another interface is extended. Hence, no other class, enumeration, or external data type can be extended (Error Code: 0xC4A09). Violations are considered as errors (Severity: error), e.g., l.11 in Listing 4.20.

Implements

The **implements**-relations allows to implement interfaces. Because the grammar does not allow to specify such a restrictions of the implementing type, a context condition ensures that the implemented type is an interface (Error Code: 0xC4A10). An example of a model containing this kind of error (Severity: error) is shown in Listing 4.21 in ll.4-5.

CD4A

```
1 classdiagram InvalidImplements {
2   enum T { PERIODIC, ONE_TIME; }
3   abstract class A;
4   class B implements T; ✗ //0xCAA10
5   class C implements A; ✗ //0xC4A10
6 }
```

Listing 4.21: The **implements**-relation allows to implement interfaces only.

Attributes

Attributes in CD4A are only allowed for CD4A classes (Error Code: 0xC4A66), as already mentioned in Section 4.2.2. Violations of this context condition are handled as errors (Severity: error). Moreover, an attribute's type has to be either an external data type or a type defined in the model (Error Code: 0xC4A14), the name of an attribute has to start with a lower case (Error Code: 0xC4A12), and the assigned value must evaluate to the type of the attribute (Error Code: 0xC4A11). By way of example, these errors (Severity: error) are shown in Listing 4.22.

Each attribute declaration can also contain an assigned value, which is only allowed for non-derived attributes (Error Code: 0xC4A34), e.g., l.15 in Listing 4.22. The reason is that semantically a derived attribute defines that the value has to be specified in the design phase. Hence, an assignment violates (Severity: error) this meaning as the value is defined in the analysis phase.

Besides a wrong type, name, or value, attribute declarations can be invalid, in particular, when used in classes with inheritance. Assuming that an attribute is already defined in a superclass, the subclass can override the attribute but only if the type and name are equal (Error Code: 0xC4A13), e.g., l.3 and l.7 in Listing 4.23. In addition, each defined attribute has to be unique, i.e., the name has to be unique, within a class

```
1 classdiagram InvalidAttributes {                                    CD4A
2   enum A {
3     int value; ✗                          //0xC4A66
4   }
5   interface B {
6     String value; ✗                       //0xC4A66
7   }
8   abstract class C {
9     MyType attribute; ✗                   //0xC4A14
10    double Value; ✗                       //0xC4A12
11    int age = "Hello World"; ✗            //0xC4A11
12    int iValue = age + 10; ✗              //0xC4A74
13    /boolean initialized = false; ✗       //0xC4A34
14  }
15 }
```

Listing 4.22: Example of restrictions for CD4A attributes' name, value, and type.

(Error Code: 0xC4A15), and cannot be abstract (Error Code: 0xC4A52). All errors (Severity: error) are shown in Listing 4.23.

```
1 classdiagram InvalidInheritanceAttribute {                          CD4A
2   abstract class T {
3     double value;
4     abstract int age; ✗ //0xC4A52
5   }
6   class A extends T {
7     String value; ✗       //0xC4A13
8     String name;
9     int name; ✗           //0xC4A15
10  }
11 }
```

Listing 4.23: Attributes have unique names and cannot be defined multiple times.

Associations

For presentational reasons, the notion of an association's *source* and *target* is introduced. Given a unidirectional association from `A` to `B`, where `A` and `B` are arbitrary types, we denote `A` as the source and `B` as the target of the association. Note that for checking context conditions of bidirectional associations, two unidirectional associations are used.

For each association it has to be ensured that it is unique (Error Code: 0xC4A26), i.e., there are no two associations with the same source and target or name. As associations may have a name and role names, this restriction implies that the association name and each role name is unique (Error Code: 0xC4A25 and Error Code: 0xC4A27). Each association name and role name has to start with a lower case (Error Code: 0xC4A16 and Error Code: 0xC4A17). If no association name or role name is given, a default name is used, which is the target's type name in lower case. Such a default name has to be unique as well (Error Code: 0xC4A28). All these restrictions are handled as errors (Severity: error) if violated. An example is shown in Listing 4.24.

CD4A

```
1 classdiagram InvalidAssociation {
2   class T{
3     double value;
4   }
5   enum A{
6     PERIODIC,
7     ONE_TIME;
8   }
9   association assoc   T -> A;
10  association assoc   T -> A; ✗          //0xC4A26
11  association MyAssoc T -> A; ✗          //0xC4A16
12  association         T -> (value) A; ✗  //0xC4A27
13  association assoc   B -> C; ✗          //0xC4A28
14 }
```

Listing 4.24: A CD4A model showing invalid association definitions, where the names are not unique or invalid.

CD4A also restricts the set of valid association source types, as already mentioned in Section 4.2.4. In particular, the source type of an association cannot be an enumeration (Error Code: 0xC4A21). This error (Severity: error) is shown in Listing 4.25 in l.5. In addition, the association's source type cannot be an external data type (Error Code: 0xC4A21) (l.6 in Listing 4.25), because external data types cannot be modified. Hence, it is also considered as an error (Severity: error).

CD4A

```
1 import java.util.Date;
2 classdiagram InvalidSourceAssociation {
3   enum T { PERIODIC, ONE_TIME; }
4
5   association T    -> A; ✗ //0xC4A21
6   association Date -> A; ✗ //0xC4A21
7 }
```

Listing 4.25: An example showing that an association's source cannot be an external data type or an enumeration.

Besides restrictions to the associations names and types, the cardinalities are restricted in some cases. One case is that whenever the «ordered»-stereotype is used, the cardinality on the same side has to be either `[*]` or `[1..*]` (Error Code: 0xC4A24), because otherwise it does not have any semantic meaning, as shown in Listing 4.26 ll.2-3. However, violations of this context condition are considered as warnings (Severity: warning), because the stereotype can be ignored. The other case is the cardinality for compositions, which is restricted to `[1]` on the whole composition side (Error Code: 0xC4A18), i.e., a part belongs to one whole (cf. [Rum12]), as shown in Listing 4.26 in ll.4-6. Violating this restriction are handled as errors (Severity: error).

```
1 classdiagram InvalidCardinalities {                                    CD4A
2   association         A  -> T [1]    <<ordered>>;  //0xC4A24
3   association         A  -> B [0..1] <<ordered>>;  //0xC4A24
4   composition [*]     C <-> D;                     //0xC4A18
5   composition [1..*] D <-> C;                      //0xC4A18
6   composition [0..1] D <-> C;                      //0xC4A18
7 }
```

Listing 4.26: The cardinality for ordered associations should be `[*]` or `[1..*]` but the cardinality of composition's whole has to be `[1]`.

For qualified associations, cardinalities are not restricted but additional restrictions are added regarding the qualifier. First, all data types used as a qualifier types have to either be external data types or types defined in the model (Error Code: 0xC4A19). For any other type, i.e., undefined types, an error is produced (Severity: error), e.g. l.6 in Listing 4.27. Second, if an attribute of the association's target type is used, it has to be present in the target type or in one of its super classes (Error Code: 0xC4A20). If it is not present, qualification is not possible and the model contains an error (Severity: error) as shown in l.7. Third, a qualified unidirectional association is correctly defined, if the qualifier type is on the side of the association's source (Error Code: 0xC4A35). Otherwise, there is no semantic meaning to the qualified association and it is considered as an error (Severity: error), e.g., l.8 in Listing 4.27.

```
1 classdiagram InvalidQualifiers {                                       CD4A
2   abstract class A {
3     long number;
4   }
5
6   association           B [MyType]   -> A;          //0xC4A19
7   association undefined B [[value]] -> A;          //0xC4A20
8   association wrongSide B           -> [number] A; //0xC4A35
9 }
```

Listing 4.27: The qualifier of a qualified association has to be either an external data type or a type defined in the model.

Types

In general, every type defined in a model or external types imported can be used within a model. However, CD4A restricts the use of generic types such as `List` and `Optional`. First, generic types are not allowed to be nested (Error Code: 0xC4A29), as shown in Listing 4.28 in l.3. This design decision is based on existing equivalent representation of such data structures by using appropriate classes and associations and, hence, considered as an error (Severity: error). Moreover, when defining generic parameters, the generic type parameter has to be explicitly defined either within the model or imported (Error Code: 0xC4A30), as shown in l.4 in Listing 4.28. Also the number of generic parameters has to be exactly one (Error Code: 0xC4A31), e.g., l.5 in Listing 4.28. Both of these context conditions are considered as errors (Severity: error).

```
1 classdiagram InvalidGenerics {                                    CD4A
2   abstract class A {
3     List<Set<String>> values;  ✗        //0xC4A29
4     List<MyType> balance;  ✗            //0xC4A30
5     List<Double,Integer> overdraft;  ✗  //0xC4A31
6   }
7 }
```

Listing 4.28: Generics in CD4A are only simple and with only one parameter.

4.3 CD4Code: Modeling Language for Implementation Models

The CD4A ML is designed for analysis models only. It lacks language constructs to be usable for modeling design and implementation models (cf. Section 4.1). However, a representation of implementation models is necessary for the adaptable transformation- and template-based code generation approach developed in this thesis (cf. *MR-3*), as described in detail in Chapter 8. Hence, in this thesis the UML class diagram for code (CD4Code) ML has been developed as a ML for design and implementation models. It is based on the CD4A ML but extends it with methods, constructors, visibility, and modifiers. Technically, CD4Code is a language extension of CD4A using MC language extension mechanism (cf. Section 2.2.1) and additional context conditions.

In summary, CD4Code supports: classes, interfaces, enumerations, attributes, external data types, constructors, methods, visibility, and abstract modifier. Moreover, CD4Code explicitly omits method and constructor implementations.

In the remainder of this section, the CD4Code ML is described as the extension of the CD4A ML. Therefore, only the CD4Code ML's extensions to the CD4A ML are introduced. Hence, this section is intended for generator developers using the CD4Code

ML's AST rather then modelers using the concrete syntax. Nevertheless, the full grammar describing CD4Code is shown in Section C.1. In addition, context conditions for CD4Code are listed in Section E.2.

4.3.1 Modifiers

In general, modifiers are markers for language constructs to change their meaning. The visibility modifier defines how the concept is viewed from within and from outside of the enclosing scope (cf. Section 2.2.3). For example, by defining an attribute to be private it is only visible within the class. CD4Code uses the four common types of visibility: private, protected, public, and default, as shown in ll.2-4 in Listing 4.29.

```
1 Modifier = Stereotype?                                          MCG
2   (  ["private"] | [private:"-"]
3    | ["protected"] | [protected:"#"]
4    | ["public"] | [public:"+"]
5    | ["abstract"] | ["final"] | ["static"]
6    | ["derived"] | [derived:"/"]
7   )*;
8
9 Stereotype = "<<" values:(StereoValue || ",")+ ">>";
10 StereoValue = Name ("=" value:String)?;
```

Listing 4.29: Modifiers allow to define stereotypes and visibilities for classes, interfaces, enumerations, attributes, methods, and constructors.

Additionally, abstract, final, static, and derived modifiers (ll.5-6) as well as stereotypes (l.1) can be used with the same meaning as defined in UML/P CD [Sch12]. Note that each stereotype is a key-value pair enclosed in «», as shown in ll.9-10.

Modifiers can be used for every CD4Code class, attribute, interface, enumeration, method, constructor, and association end. However, CD4Code classes, interfaces, enumerations, methods, and constructors cannot be derived. Moreover, interfaces, enumerations, associations, attributes, and constructors cannot be abstract. Neither interfaces nor enumerations can have private or protected visibility, or be final or static. Association ends can have visibilities and arbitrary stereotypes. However, only one visibility per modifier is allowed.

4.3.2 Constructor-Signatures

For every class, a constructor allows to create an instance of the enclosing class. In the implementation of the CD4Code ML, constructors with varying parameters are supported as shown in l.2 in Listing 4.30. A constructor is only allowed in a class, and an enumeration. However, no implementation bodies are allowed (cf. Section 4.1).

```
1 CDConstructor = Modifier Name                                    MCG
2   "(" (CDParameter || ",")* ")"
3   ("throws" exception:(QualifiedName || ",")+ )?
4   ";";
```

Listing 4.30: Each CD4Code class and enumeration can define a constructor with a set of parameters but no implementation body.

Since this thesis assumes the target language to be Java, CD4Code additionally supports throws declarations to denote exceptions thrown by methods (l.3 in Listing 4.30).

4.3.3 Method-Signatures

Besides constructors, methods are supported in CD4Code as well, each of which has one return type, a name, and a list of parameters, as shown in Listing 4.31. In compliance to constructors, no implementation bodies are supported. In addition, a method signature may be extended with an additional exception declaration that is thrown by this method (l.3 in Listing 4.31).

```
1 CDMethod = Modifier ReturnType Name                              MCG
2   "(" (CDParameter || ",")* ")"
3   ("throws" exception:(QualifiedName || ",")+ )?
4   ";";
```

Listing 4.31: CD4Code supports method-signatures with modifiers, return types, and parameters but no implementation bodies.

In general, CD4Code supports methods for CD4Code interfaces and enumerations.

4.3.4 CD4Code Interface

In contrast to the CD4A ML, interfaces in the CD4Code ML are not primarily markers. Interfaces may extend other interfaces, provide method-signatures, and are allowed to define attributes, as shown in Listing 4.32. Each defined attribute has to be static.

```
1 CDInterface astimplements ASTCDType =                            MCG
2   Modifier? "interface" Name
3   ( "extends" interfaces:(ReferenceType || ",")+ )?
4   ( "{" ( CDAttribute | CDMethod )* "}" | ";" );
```

Listing 4.32: CD4Code supports methods and static attributes in interfaces.

4.3.5 CD4Code Enumeration

Due to the focus of this thesis to generate Java source code, enumerations in the CD4Code ML are regarded similar to Java enumerations. In particular, enumerations are allowed to implement interfaces defined in the CD4Code model, as shown in l.2 in Listing 4.33. Besides enumeration constants, CD4Code enumerations can have arbitrary many methods as well as constructors (l.4).

```
1 CDEnum astimplements ASTCDType = Modifier? "enum" Name                MCG
2   ( "implements" interfaces:(ReferenceType || ",")+ )?
3   ( "{"
4       (CDEnumConstant || ",")* ";" (CDConstructor | CDMethod)*
5     "}"
6   | ";" );
```

Listing 4.33: CD4Code supports methods and constructors in enumerations.

Chapter 5

Systematic CD4A ML to a Java Mapping

The CD4A ML, which has been introduced in Chapter 4, is the primary development artifact to specify the structured data managed by a data-centric application. In MDP and MDD of data-centric applications, as described in Chapter 3, CD4A models are mapped to a data structure, which is the source-code-representation of CD4A models.

In general, a mapping depends on (i) the source language, (ii) the target language, (iii) how the generated source code is used by the varying roles (cf. Section 3.1.3), and (iv) additional mapping guidelines. In this thesis, a systematic mapping of the CD4A ML to the Java source code has been developed (cf. *RE-5*). This mapping, which is a more detailed extension of the mappings introduced in [Rum12, BFL13], ensures data consistency (cf. *RE-6*), i.e, all mandatory associations (associations with cardinality `[1]` and `[1..*]`) specified in a CD4A model have at least one association link, and all attributes are assigned with a valid value.

The goal of this chapter is to introduce the developed CD4A-to-Java mapping to generator developers. This is done based on the example presented in Chapter 4. Hence, this chapter is structured as follows. First, the general considerations and the used mapping guidelines are presented in Section 5.1. Afterwards, the mapping for each CD4A language concept to Java language concepts is explained in Section 5.2. Because this mapping ensures consistency, data structures resulting from CD4A models with mandatory-to-mandatory associations cannot be instantiated. Hence, approaches to overcome this limitation are discussed in Section 5.3.

5.1 General Considerations and Mapping Guidelines

Each mapping requires guidelines it relies on such that every ambiguity of the mapping is resolved relative to the guidelines (cf. *RE-5-1*). For example, how the names of variables and methods in the Java source code are chosen depends on whether understandability and usage of the generated source code is a goal. The mapping used in this thesis follows the subsequently listed guidelines.

- **Systematic Derivation:** A mapping of CD4A language concepts to Java source code has to be defined in a systematic and very specific way, i.e., besides defining

the Java source code resulting from a particular CD4A language concept, it has to define how this mapping is realized. For example, by describing how the name and type of a Java variable are chosen. No ambiguity or incompleteness, i.e., an unspecified mapping of a CD4A language concept, is allowed, because the code generator requires clear instructions to systematically generate Java source code.

Such a systematic mapping is also essential for the use of the generated Java source code. If the mapping is understood, inspection of the generated source code is not required but interfaces are sufficient.

- **Unified Use of Wording:** A unified use of wording used in a CD4A model and the wording used in the Java source code is helpful to reduce a potential conceptual gap between the CD4A model and the generated Java source code. It has to be removed in order to keep traceability and promote name recognition [HBR00]. Consequently, the name schema used in the Java source code, e.g., class, variable, and method name, has to be derived from elements of the CD4A model.

- **Information Hiding and Encapsulation:** Two well-known principles in object-oriented programming are the information hiding and encapsulation principle [Par71, Par72]. The primer postulates the idea to restrict access to the implementation rather than forcing a developer to understand it. In the developed mapping, this principle is realized by providing mutators and accessors and separating the behavior from its representation using high-level interfaces. Hence, the generated Java source code can be used by only regarding the generated interfaces (cf. RE-5-1-1).

  The encapsulation principle describes the encapsulation of the object's state and behavior in the implementation. This is especially relevant when mapping CD4A associations, because each association influences the state of the linked object(s). As a result, association links and the association's behavior, i.e., methods to manage associations, are located in the association ends rather than in an association class as, e.g., proposed in [Ges08, Ges09, GW11].

- **Association Consistency:** In general, the different purposes of an association in UML CD converge to "some kind of dependency between two classes" [Ste02]. A relationship between two classes `A` and `B`, where `A` is dependent on `B`, refers to the possible change of an instance of `A` if an instance of `B` is changed. This relationship understanding cannot directly be mapped to Java, because a similar concept does not exist (cf. [Ges09]).

  A Java implementation that realizes the dependency between two UML CD classes has to ensure associations consistency for bidirectional associations (also called "referential integrity" [BFL13]), i.e., the correct realization of the mutual update in the dependency relation. A plethora of approaches to ensure association consistency have been proposed, e.g., [HBR00, Ste02, GDCL03, MZ04, AHMM07,

Ges08, Ges09, GW11, Rum12], each of which varies conceptually and technically. The developed in this thesis extends already existing non-reflective and lightweight approaches [Rum12, BFL13] that address association cardinality constraints and prevents violations through external method calls.

- **Static Type-safety:** Type-safety can either be ensured dynamically or statically [Har12]. Dynamically ensuring type-safety requires detection of the object's type at run-time and may result in unpredictable application behavior that is not visible a priori. In contrast, static type-safety detects type inconsistencies at compile time and, hence, omits such runtime errors.

  Type-safety is especially relevant for associations. For example, a dynamic typing such as proposed in [GDCL03, TM05, Ges09] may lead to unwanted run-time exceptions because reflective access is required to deduce an object's type. The proposed mapping aims for static type-safety and explicitly avoids any dynamic casting at runtime.

- **Resolve Semantic Variation Points:** CD4A harbors semantic variation points [Grö10] (cf. Section 1.2). Yet, such a degree of freedom is obstructive for code generation, because a code generator requires clear rules to systematically map each concept of the analysis model to concepts of the target language. To resolve this issue, a default mapping for every semantic variation point has to be designed.

Note that thread-safety is not handled due to the focus on client applications that only support manipulation of one object at a time. Such concerns can be added by either Java constructs as proposed in [Rum12] or atomic action concepts [Cac07].

5.2 Mapping of CD4A Concepts to Java Source Code

The CD4A-to-Java mapping used in this thesis is mainly based on [Rum12, BFL13] and adapts it to the general considerations and mapping guidelines listed in Section 5.1. The mapping is presented for each CD4A language concept by use of examples. For presentational purposes, we use the example introduced in Section 4.2 (Figure 4.1 on page 48) and additional examples to discuss relevant details. Moreover, to refer to the Java language concepts mapped from a CD4A language concept, we use the wording: e.g., Java interface of the CD4A class. In addition, whenever appropriate existing mappings are discussed in detail.

5.2.1 Mapping CD4A Model Definition

A CD4A model definition contains a package name, imports, and a model's name (Section 4.1.1). The package name and the CD4A model name are used for structuring the

generated Java artifacts. For example, in Listing 4.1 on page 49 the package `dex` and the model name `BankingSystem` are defined. The structuring, i.e., package structure of the generated Java source code, used in this thesis uses both names in lower case, e.g., `dex.bankingsystem`, to ensure that multiple models using the same package are located in different sub packages. Note that this mapping is sufficient for processing single CD4A models. Hence, if multiple CD4A models are processed and used for code generation, an alternative approach may be required. An overview of such alternative approaches is discussed in [Sch12].

Each CD4A import is mapped to the equivalent Java `import` concept for each generated artifact as well, because the imports in the CD4A model are defined globally, i.e., for all enclosed classes and interfaces. Imports of external CD4A types are not supported.

5.2.2 Mapping CD4A Interfaces

A CD4A interface is mapped to a public Java interface. An example of this mapping is shown in Figure 5.1. The CD4A interface `Employee` (at the top) is mapped to a public Java interface with the same name (at the bottom).

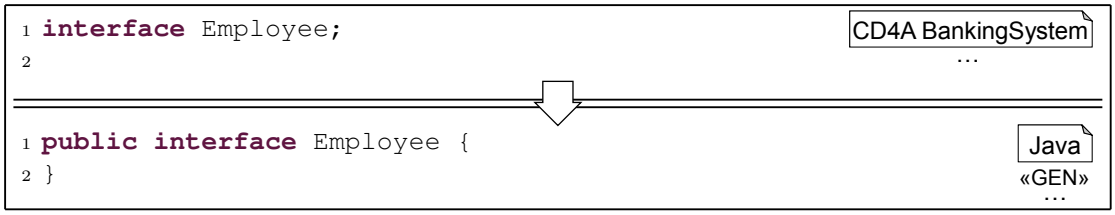


Figure 5.1: An example of mapping a CD4A interface (at the top) to a Java interface (at the bottom).

If the CD4A interface has outgoing associations, the mapping adds additional method declarations in the Java interface. Note that a detailed explanation of the mapping for CD4A associations is described in the remainder of this chapter. The implementation of the method declarations is regarded by the mapping of the CD4A classes implementing the CD4A interface. A Java implementation of the CD4A interface does not exist, because it leads to situations, where multi-inheritance is required in the Java source code. For example, the model shown in Listing 5.1 requires multi-inheritance, when mapping the interface `A` (l.1), `B` (l.2), and `C` (l.3) to Java interfaces and implementations.

```
1 interface A;                                                    CD4A
2 interface B;
3 interface C extends A, B;
```

Listing 5.1: A CD4A model that requires resolving multi-inheritance when mapping CD4A interfaces to Java interfaces and implementations.

A CD4A interface extending another CD4A interface is mapped to a Java interface that uses the Java `extends` language concept to extend the Java interface mapped from the extended CD4A interface. For instance, the CD4A interface `B` extends the CD4A interface `A`, as shown in Figure 5.2 (at the top). The Java interface for the CD4A interface `A` is shown at the bottom left. The Java interface of the CD4A interface `B`, which is shown at the bottom right, extends the Java interface `A`. If the CD4A interface extends multiple other CD4A interfaces, the Java interface uses a comma separated list of extended Java interfaces.

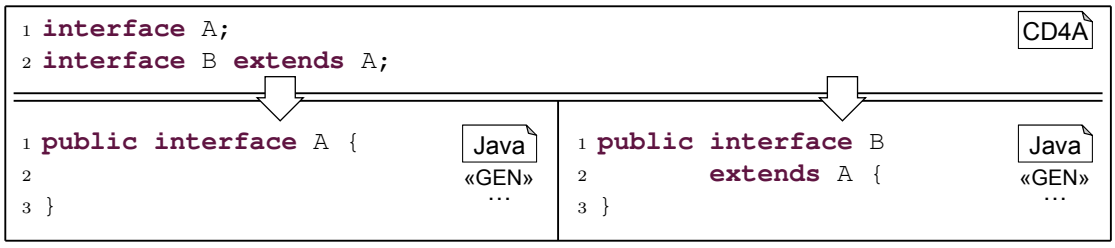


Figure 5.2: An example of mapping CD4A interface's `extends`-concept (at the top) to Java's `extends`-concept (at the bottom).

5.2.3 Mapping CD4A Classes

A CD4A class is mapped to a public Java interface and a public Java class, which implements the Java interface. The Java interface's name is the name of the CD4A class. The Java class' name is composed of the CD4A class' name and a `Impl`-suffix. The Java interface and class are required to separate the representation from the implementation (cf. Section 5.1).

The example in Figure 5.3 shows the CD4A class `A` (at the top) that is mapped to the public Java interface `A` (bottom left) and the public Java class `AImpl` (bottom right) implementing the `A` interface.

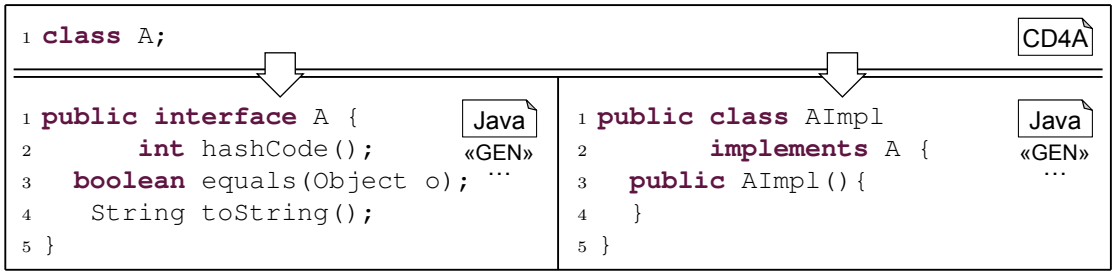


Figure 5.3: An example of a CD4A class (`A` at the top) that is mapped to a Java interface (`A` bottom left) and a Java implementing class (`AImpl` bottom right) with additional standard Java methods.

In addition, the example shows that by default the Java interface contains an `equals()`, a `hashCode()`, and a `toString()` method for comparing objects and converting an object to a `String` value, as proposed in [Rum12]. All methods are implemented in the Java class, e.g., the `AImpl` Java class. For further details on their implementation, interested readers are advised to consider [Blo08].

Alternative approaches to map a UML CD class to an interface and an implementation have been proposed. Harrison et. al. proposes to make the implementation abstract and add an additional implementing subclass in order to use the Generation Gap-Pattern [HBR00]. A similar approach has been proposed in [MSHL06]. The Eclipse Modeling Framework (EMF) [SBPM09] maps a UML CD interface to a Java interface and a Java implementation but additionally adds infrastructure related implementations.

Abstract CD4A Classes

An abstract CD4A class is mapped to a Java interface and an abstract Java class. For example, the abstract CD4A class `A` (at the top in Figure 5.4) is mapped to the Java interface `A` (bottom left) and the abstract Java class `AImpl` (bottom right).

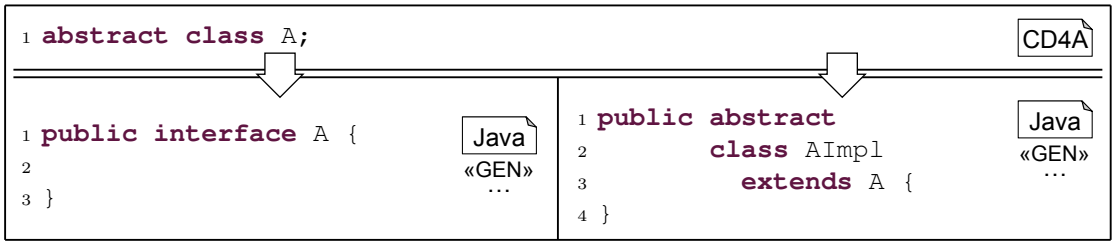


Figure 5.4: An abstract CD4A class (`A` at the top) that is mapped to a Java interface (`A` bottom left) and a Java implementing class (`AImpl` bottom right).

An implication of this mapping is that if a CD4A model contains only abstract classes, the Java source code cannot be instantiated, i.e., no object of a Java class can be created. However, such a data structure may serve a reuse purpose. Alternatively, the CD4A model can be restricted to only allow abstract classes with at least one concrete subclass, as proposed in [Lan16].

CD4A Class Hierarchies

The CD4A ML allows that CD4A classes implement CD4A interfaces or extend other CD4A classes. Whenever a CD4A class implements a CD4A interface, the Java class mapped from the CD4A class implements, i.e., using the Java `implements` concept, the Java interface mapped from the CD4A interface. If a CD4A class extends another CD4A classes, the following mapping is used:

- The Java interface of the CD4A subclass extends the Java interface of the CD4A superclass.
- The Java class of the CD4A subclass implements the Java class of the CD4A superclass.
- The Java class' constructor of the CD4A subclass contains the CD4A superclass' attributes and mandatory associations.

To demonstrate the mapping, an excerpt of the banking system example in Figure 4.1 is used, which is depicted in Figure 5.5. In this example, the CD4A class `CheckingAccount` extends the CD4A class `Account` (l.1 at the top). The generated Java interface `CheckingAccount` extends the Java interface `Account`, which is the generated interface for the CD4A class `Account` (ll.2-3 at the bottom left). The generated Java class `CheckingAccountImpl`, implements the `Account` Java interface (l.4 at bottom right) by extending the Java implementation `AccountImpl` to reuse the implementation of its superclass (l.3 at bottom right).

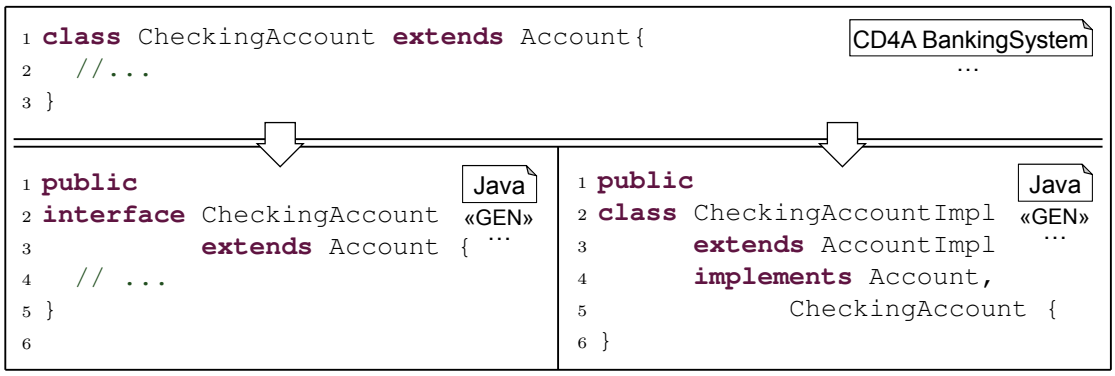


Figure 5.5: An example of a CD4A class (`CheckingAccount` at the top) with an `extends`-relation that is regarded in the Java source code (bottom right and bottom left).

5.2.4 Mapping CD4A Enumerations

A CD4A enumeration is mapped to a public Java `enum` with the same name as the CD4A enumeration. The Java enumeration's values are the CD4A enumeration's values. To illustrate the mapping, the example shown in Figure 5.6 depicts the CD4A enum `TransactionType` (at the top) that is mapped to a Java enumeration `TransactionType` (at the bottom).

An alternative mapping approach is to use a Java interface and `public static final` variables to represent each constant. However, each constant has to be handled

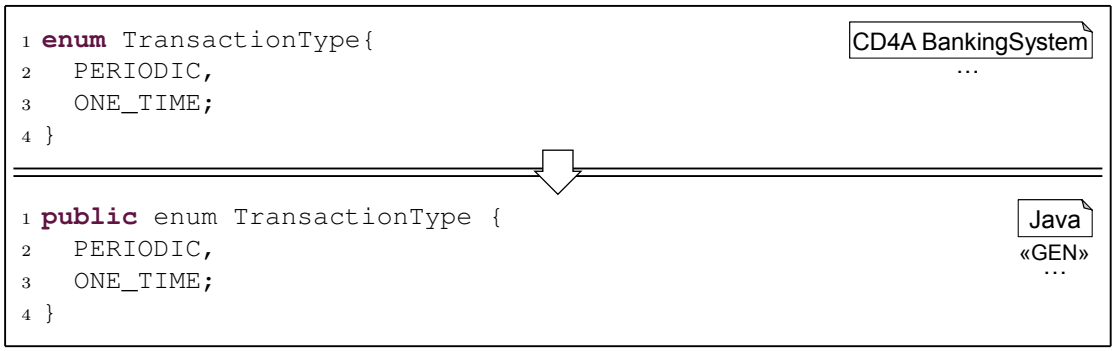


Figure 5.6: A CD4A enumeration (`TransactionType` at the top) is mapped to a Java enumeration (`TransactionType` at the bottom) with the same name.

as an `int` and, hence, type-safety for different enumeration types is not guaranteed. Another approach is by using the *Typesafe Enum*-Pattern [Blo08], as proposed in [SBPM09]. This mapping has been proposed due to technical limitations of Java 1.5, because it was the only way to implement a type that can take a finite number of values. Among others, it's disadvantage is that its constants cannot be used in `switch` statements.

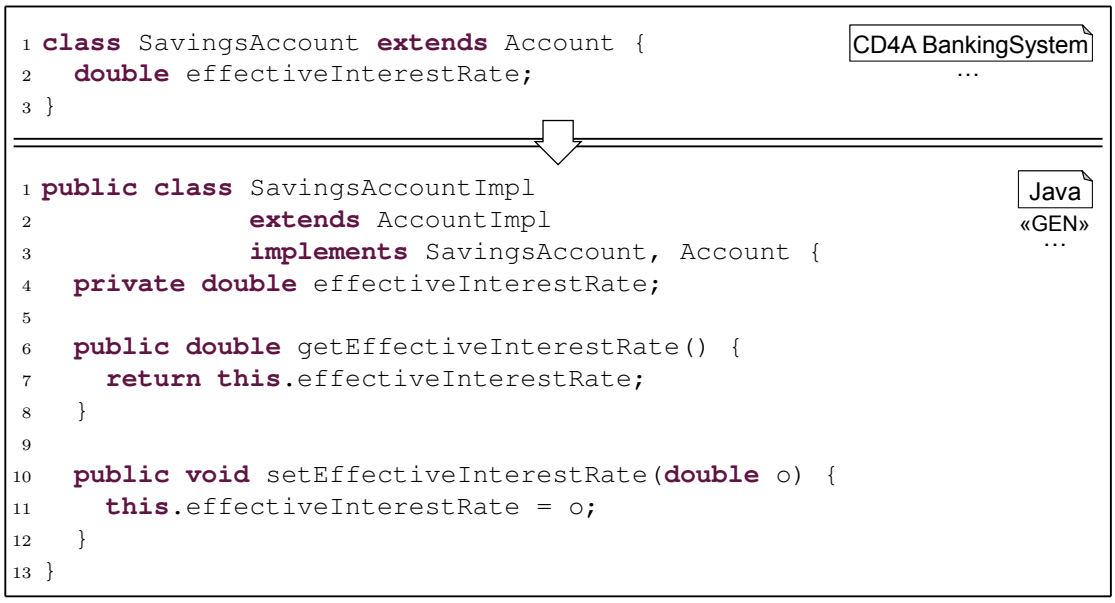
5.2.5 Mapping CD4A Attributes

Each CD4A class can contain multiple attributes, each of which is mapped to a private Java variable within the Java class mapped from the CD4A class. Private visibility is used to respect the information hiding principle (cf. Section 5.1). Additionally, a CD4A attribute is mapped to a public accessor and a public mutator. Note that this is a common mapping, e.g., [HBR00, SBPM09, Rum12].

The Java variable's type is the CD4A attribute's type. However, if it is a primitive generic type (e.g., `Optional<int>`), it is mapped to a generic wrapper type (e.g., `Optional<Integer>`). The variable's name is the CD4A attribute's name. Moreover, if default values assigned to a CD4A attribute, are kept. For example, some mappings of CD4A attributes to Java variables are shown in Table 5.1. The CD4A attributes on the left are mapped to the Java variables on the right, e.g., the `List<Double> c` CD4A attribute is mapped to the `private List<Double> c` Java variable.

CD4A Attribute	**Java Variable**
`int a;`	`private int a;`
`Optional<int> b;`	`private Optional<Integer> b;`
`List<Double> c;`	`private List<Double> c;`
`E e;` (where `E` is a CD4A class, interface, or enumeration; or an external data type)	`private E e;`

Table 5.1: An example of the mapping of CD4A attributes to Java variables.




```
1 class SavingsAccount extends Account {
2   double effectiveInterestRate;
3 }
```

```
1 public class SavingsAccountImpl
2              extends AccountImpl
3              implements SavingsAccount, Account {
4   private double effectiveInterestRate;
5
6   public double getEffectiveInterestRate() {
7     return this.effectiveInterestRate;
8   }
9
10  public void setEffectiveInterestRate(double o) {
11    this.effectiveInterestRate = o;
12  }
13 }
```

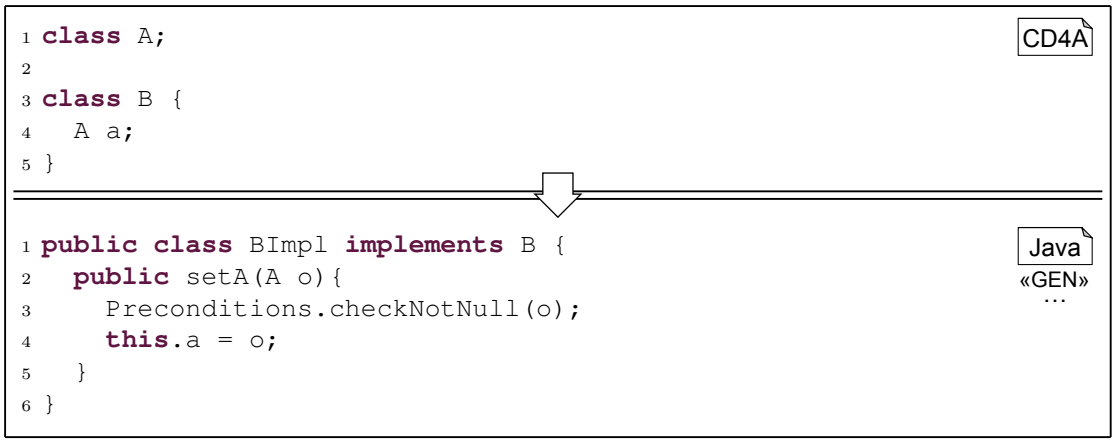
Figure 5.7: The CD4A attribute (l.2 at the top) is mapped to a private Java variable (l.4 at the bottom), and an accessors and a mutator (ll.6-12 at the bottom).

The CD4A attribute's name is also used to derive the names of the accessor and mutator for accessing and manipulating the Java variable. The accessor's name is composed of a `get`-prefix and the capitalized CD4A's attribute name. For conventional reasons, the accessor has a `is`-prefix, if its type is boolean. The mutator's name is composed of a `set`-prefix and the attribute's name.

For example, Figure 5.7 shows the mapping for the `effectiveInterestRate` CD4A attribute (l.2 at the top), which is mapped to the private Java variable `effectiveInterestRate` (l.4 at the bottom) and two public methods (ll.6-12 at the bottom). The `getEffectiveInterestRate()`-method (ll.6-8 at the bottom) is the accessor and returns the Java variable's value. The mutator (`setEffectiveInterestRate()`-method in ll.10-12 at the bottom) has one parameter with the Java variable's type and the name `o`.

The implementation of the mutator changes, if the CD4A attribute's type is not primitive. In this case, the mutator checks whether the value is not `null` and may throw an exception to ensure data consistency. For example, the `getA()`-method in ll.2-5 in Figure 5.8 does a precondition check (l.3) before assigning the value (l.4).

In general, attributes with the same name and type may occur multiple times in a hierarchy of CD4A classes. In this case, the child attributes have to be removed to avoid run-time exceptions when mapping to Java source code. The run-time exception occurs during instantiation of the subclass, when the duplicate variable values are set. In this case, only the subclass variable will be set but not the superclass variable because the




```
1 class A;
2
3 class B {
4   A a;
5 }
```

```
1 public class BImpl implements B {
2   public setA(A o){
3     Preconditions.checkNotNull(o);
4     this.a = o;
5   }
6 }
```

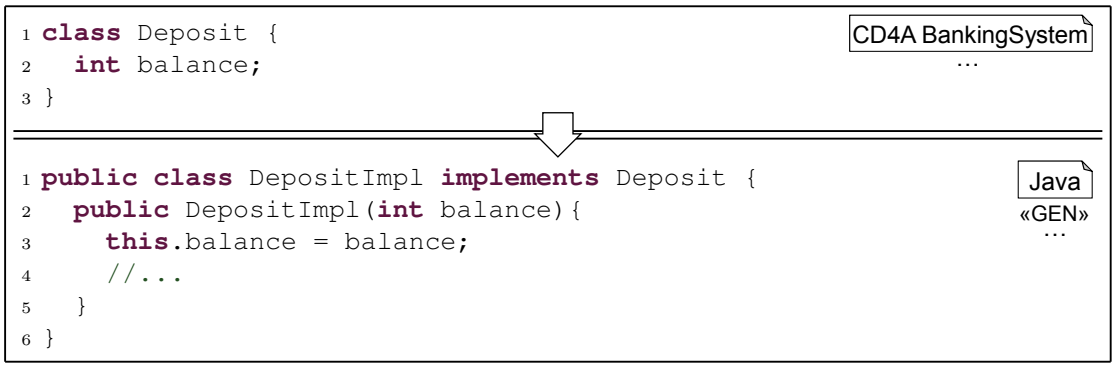
Figure 5.8: An example of a mutator for non-primitive CD4A attributes.

mutator is overridden. In consequence, calling the super constructor will raise a run-time exception, because its variable is `null` (violation of data consistency).

Ensuring Data Consistency in Constructors

A CD4A class without attributes is mapped to a Java class with a default constructor. If a CD4A class contains attributes, it has to be ensured that when an instance of the Java class is created, a value is assigned to the Java variable. Hence, no default constructor exists. Such a mapping has also been proposed in [HBR00, Rum12, BFL13, Lan16].

In particular, each CD4A class with attributes is mapped to a Java class with a constructor that has a parameter for each CD4A class' attribute. An example is shown in Figure 5.9. In this example, the CD4A class `Deposit` (at the top) is mapped to the Java class `DepositImpl` (at the bottom) with a constructor that has one parameter for the attribute `balance` (l.2 at the bottom). The constructor assigns the value to the Java variable `balance`. If the attribute has a different type other than primitive, the constructor needs to check if the value is not `null`.




```
1 class Deposit {
2   int balance;
3 }
```

```
1 public class DepositImpl implements Deposit {
2   public DepositImpl(int balance){
3     this.balance = balance;
4     //...
5   }
6 }
```

Figure 5.9: An example of mapping a CD4A class (at the top) to a Java class (at the bottom) that ensures data consistency in the constructor.

Derived CD4A Attributes

Derived CD4A attributes are mapped to a Java accessor only and no Java variable or mutator, because derived attributes cannot be modified (cf. [Rum12]). Hence, derived attributes are not considered in constructors of Java classes.

The accessor uses the same naming schema as accessors for non-derived attributes. By default the accessor throws an exception to denote a missing implementation. Alternatively, it is also possible to return default values instead throwing an exception. However, default values can only be returned for primitive derived attributes. Moreover, default values introduces a conceptual gap, because the application developer is not aware of the missing implementation.

To demonstrate the mapping the example in Figure 5.10 is considered. It shows the mapping for the derived CD4A attribute `completed` (l.1 at the top). It is mapped to the `isCompleted()` accessor with public visibility that throws a `NotImplemented-Exception` (shown in ll.2-4).

CD4A BankingSystem
...

```
1 class Transaction {
2   /boolean completed;
3   //...
4 }
```

Java
«GEN»
...

```
1 public class TransactionImpl implements Transaction {
2   public boolean isCompleted() {
3     throw new NotImplementedException("0xD5078: ...");
4   }
5 }
```

Figure 5.10: The derived CD4A attribute `completed` (l.1 at the top) is mapped to the `isCompleted()` accessor (ll.2-4 at the bottom).

5.2.6 Mapping CD4A Unidirectional Associations

Associations in CD4A are based from the old and intuitive notion of a relationship between entities in Entity-Relationship (ER) modeling [Mil07]. They "have to be implemented by an adequate combination of classes, attributes, and methods" [GDCL03], where adequate depends on general considerations as explained in Section 5.1. Hence, different approaches of mapping UML CD associations to object-oriented source code have been discussed, e.g., [Fow97, KNNZ99, Ste02, DD06, Mil07, Øs07, WS07, DED08, SBPM09, Rum12, Ste13, BFL13, Lan16]. Each mapping addresses a different area of interest such as visibility of association ends, thread-safety, or understandability. Even object-oriented programming languages have been proposed to handle associations in order to improve maintenance [Ste13].

The focus of this thesis is on *static association*, i.e., the association's implementation in the software system can be identified by inspecting the code [Ste02], because the "linked objects are conceptually *aware* of the relationship" [BFL13]. Each association link represents a structural link between instances that last longer than only during an interaction between instances [Mil07]. Furthermore, associations are restricted to bidirectional associations without visibility and ownership. Approaches to handle n-ary associations with visibility do exist, e.g., [DD06, DED08, Ges09].

The developed mapping for associations is rooted on and extends [Rum12, BFL13]. It requires (i) a Java variable with an appropriate type to store the association links and (ii) Java methods to manipulate and manage associations links. In addition, to ensure data consistency (iii) the constructor and mutators may have to respect mandatory associations (cf. Section 5.2.3). Moreover, the association's implementation, i.e., Java variable and Java methods, is mapped to the Java class of the association's source (cf. Section 4.2.4). Hence, each association's source (cf. Section 4.2.4) is responsible for managing the association's links.

The name of the Java variable, which stores association links, is derived using the following priority list: (1) role name, (2) association name, (3) lower case association target. Its type depends on the kind of association and the cardinality. For example, in Table 5.2 the mapping of ordinary associations to Java variables is shown. The Java variable's type is the association's target type for associations with cardinality `[1]`. The optionality of `[0..1]` associations is represented by the `Optional` type to neglect `null`-values (this, among others, is a separating criteria to [BFL13]). Furthermore, the `Set` type containing the association's targets is used for associations with cardinality `[1..*]` and `[*]`. Note that the variables are private to respect information hiding (cf. Section 5.1). For presentational reasons, the **`association`** keyword is omitted.

CD4A Association	**Java Variable**
`A -> B [1]`	`private B b;`
`A -> (role) B [0..1]`	`private Optional<B> role;`
`name A -> B [*]`	`private Set<B> name;`
`name A -> (role) B [1..*]`	`private Set<B> role;`

Table 5.2: An example of how the Java variable to manage association links depends on the cardinality and the existence of an association name or a role name.

The Java methods to modify the Java variable storing the association link(s) also depend on the association's cardinality and the association's type. For example, associations with no modifiers and cardinality `[1]` or `[0..1]`, are mapped to a public mutator and a public accessor as shown in Figure 5.11. The mutator's name is composed from a `get`-prefix and the capitalized name of the association, e.g., `setType()` in l.2 at the

bottom. The mutator's parameter has the association target's type and the association's name, e.g., `TransactionType type` in l.2 at the bottom. The accessor's name consists of a `set`-prefix and the name of the association, e.g., `getType()` in l.3 at the bottom. Its return type is the type of the association target, e.g., `TransactionType` in l.3 at the bottom.

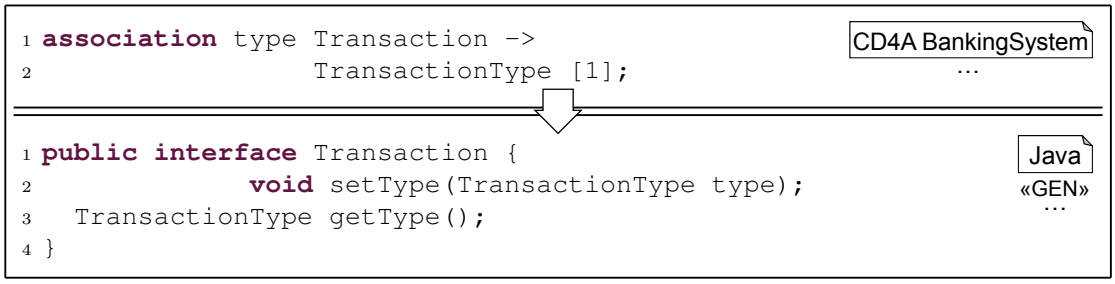


Figure 5.11: The CD4A association at the top is mapped to the Java accessor `getType()` and the mutator `setType()`.

Another mapping example for association with cardinality `[0..1]` is shown in Figure 5.12. While the accessors and mutators in l.2 and l.3 (at the bottom) are the same as for association with cardinality `[1]` but with an `Optional` data type, an additional mutator in l.3 (at the bottom) is added, which wraps the passed value into an `Optional` and calls the mutator in l.2 (at the bottom).

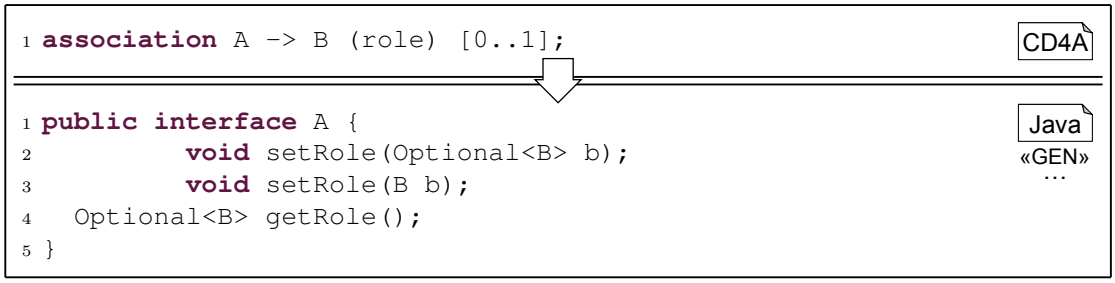


Figure 5.12: For CD4A association with cardinality `[0..1]` an additional mutator having the enclosed generic type is added.

Besides accessors and mutators, methods to add association links, remove association links, retrieve the size of association links, and check for existence of an association link are present for associations with `[*]` or `[1..*]`. In addition, for each mutator, a second mutator, which have the s-suffix, with a collection of association links as a parameter exists. Moreover, an iterator to allow traversal of the association links exist. For example, the association in Figure 5.13 (at the top) is mapped to the Java methods shown in ll.2-12 (at the bottom). Note that accessors and mutators in this case are the methods in ll.2-3 and l.11 (at the bottom), which handle a collection of association links.

This example also shows that the naming schema, as proposed above, is used for all Java methods. Furthermore, the visibility of all methods is public.

CD4A

```
association A -> B [*];
```

Java «GEN» ...

```
public interface A {
        boolean addB(B o);
        boolean addAllBs(Collection<B> o);
        boolean containsB(B o);
        boolean containsAllBs(Collection<B> o);
        boolean removeB(B o);
        boolean removeAllBs(Collection<B> o);
           void retainAllBs(Collection<B> o);
           void clearBs();
            int sizeBs();
    Iterator<B> iteratorBs();
        boolean isEmptyBs();
}
```

Figure 5.13: CD4A association with cardinality `[*]` are mapped to methods for managing sets of association links.

Technically, the returned iterator is realized as an immutable iterator, which provides a fail-fast solution for modifications. The use of iterators rather than lists has also been proposed in [SBPM09].

Not all of the Java methods shown in Figure 5.13 are necessary if the cardinality of an association is `[1..*]`. In particular, the `clearBs()`- and `isEmptyBs()`-methods are not allowed, because the collection of association links has to have at least one association link. Moreover, to ensure data consistency, the Java methods to remove and retain association links may throw a `DataStructureViolationException` exception. To avoid such errors, an additional method, which checks if there is at least one association link present and only then deletes an association link, is provided for such cases. For example, assuming the cardinality of the example in Figure 5.13 is `[1..*]`, then the `removeBIfNotLast(B o)`-method exists.

Ensuring Data Consistency in the Constructor

The constructor of a Java class that is mapped from a CD4A class that has outgoing mandatory associations, has to ensure data consistency. In this case, one of the following approaches can be used [GDCL03]:

1. Create an instance and then later set the mandatory association link.
2. Create an instance only with the mandatory association link given.

3. Create an instance and issue creation of an instance to satisfy the mandatory association link.

Approach (1) leads to a temporary inconsistent data structure, because an instance is created without the mandatory association link. Approach (3) ensures that a new instance for the association target is created, which correlates to a mapping for a composition according to [Rum12]. Finally, approach (2), which requires outgoing mandatory association links as a parameter in the constructor, conflicts with mandatory-to-mandatory associations. For example, if class `A` and class `B` are connected with a mandatory-to-mandatory association, neither an instance of class `A` nor an instance of class `B` can be created. In this thesis, approach (2) is used, because it has proven to ensure data consistency (cf. [BFL13]). Its disadvantages are tackled by appropriate methods, as described in Section 5.3.

For example, the association shown in Figure 5.14 at the top is regarded by the mapped Java class of association's source, i.e., `AImpl` at the bottom. The constructor in l.2 has one parameter for the mandatory association with the type of the Java interface mapped for the association target, i.e., `B`, and the name of the association, i.e, `b`.

CD4A

```
1 class A;
2 class B;
3
4 association A -> B [1];
```

Java

«GEN»

```
1 public class AImpl implements A {
2   public AImpl(B b) {
3     setB(b);
4   }
5 }
```

Figure 5.14: An example mapping a mandatory association (at the top) that is required to be set in the constructor (at the bottom) to ensure data consistency.

5.2.7 Mapping CD4A Bidirectional Associations

Bidirectional associations are technically implemented like two unidirectional associations -one for each navigation direction- and, hence, are mapped accordingly (cf. Section 5.2.6). In addition, association consistency, as understood in Section 5.1, is ensured based on the underlying assumption to handle it at modification-time, i.e., when an association link is added. An alternative approach would be to allow modifications and, afterwards, check constraints when the accessor is called, as proposed in [GDCL03]. However, this approach requires temporary data inconsistency.

In general, to realize association consistency, different approaches have been proposed, e.g., [Rum12] (`local()`-methods) or [MZ04, TM05] (`link()` and `unlink()`-methods). Or the use of atomic creation and updates methods has been proposed [GDCL03], which is only applicable for the simplest cases. Even the use of a centralized association class which ensures association consistency [Lan16]. A similar approach of using one class to handle associations has been proposed in [Ges09]. It reduces the synchronization challenge but also introduces an additional challenge of keeping multiplicity constraints (cf. [GDCL03]).

In this thesis, an approach based on an additional method to ensure association consistency is used. Conceptually, it is an extension of the approach proposed in [Rum12]. In particular, associations with cardinality `[1]` or `[0..1]` are mapped to additional methods to locally set the association link and locally remove it. The primer method is required to locally set the new association without introducing an infinite loop (cf. [Rum12]), whereas the latter is introduced to locally remove the association link, if no inconsistency of the data structure is the consequence. Both methods have the same return type and parameters as the association's accessors and mutators but the primer method has a `rawSet`-prefix and the latter a `rawUnset`-prefix. Note that all methods have to be public, because they are accessed from the opposite instance. However, they should never be called manually.

For associations with cardinality `[*]` and `[1..*]`, the mapping is similar but the additional methods have a `rawAdd`- and a `rawRemove`-prefix. In addition, a method with a `rawAddAll`-prefix exists to handle a collection of association links.

For instance, Figure 5.15 shows the additional methods for both association ends of the association at the top. Accessors and mutators, as described in Section 5.2.6, are omitted due to presentational reasons. For the navigation direction from `A` to `B` with cardinality `[0..1]`, the additional methods `rawSetB(Optional<B> o)` and `rawUnSetB()` exist, as shown at the bottom left. For the opposite navigation direction with cardinality `[1..*]`, the Java source code contains the additional methods `rawAddA(A o)`, `rawAddA(Collection<A> o)`, and `rawRemoveA(A o)`, as shown in ll.2-5 at the bottom right.

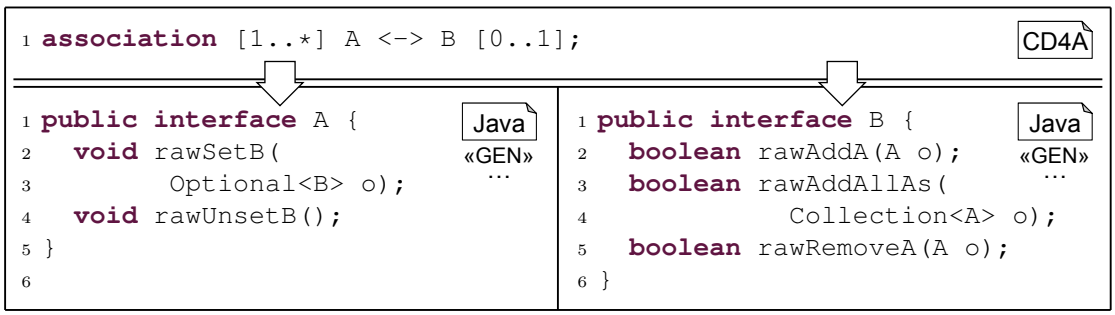


Figure 5.15: Methods provided to ensure association consistency for the association with cardinality `[0..1]` (bottom left) and cardinality `[1..*]` (bottom right).

To demonstrate how association consistency and data consistency are handled, the example in Figure 5.16, which is based on Figure 5.15, is used. In this example, it is assumed that the `setB()`-method is called. It checks if the passed value is not `null` and if currently another association link exists (`b.isPresent()`). If it does exist, then the opposite `rawRemoveA()`-method is called to remove the current association link to set a new one. Assuming it is present, the association link is simply removed by calling the `rawRemoveA()`-method of the currently linked object `b1`. Because the cardinality from `B` to `A` is `[1..*]`, the `rawRemoveA()`-method checks if there is at least one association link left and may throw an exception if necessary. Otherwise, the new association link is simply removed. Finally, the new association link is set by calling the `rawAddA()`-method, which checks if the association link to be set is not `null` and sets it locally.

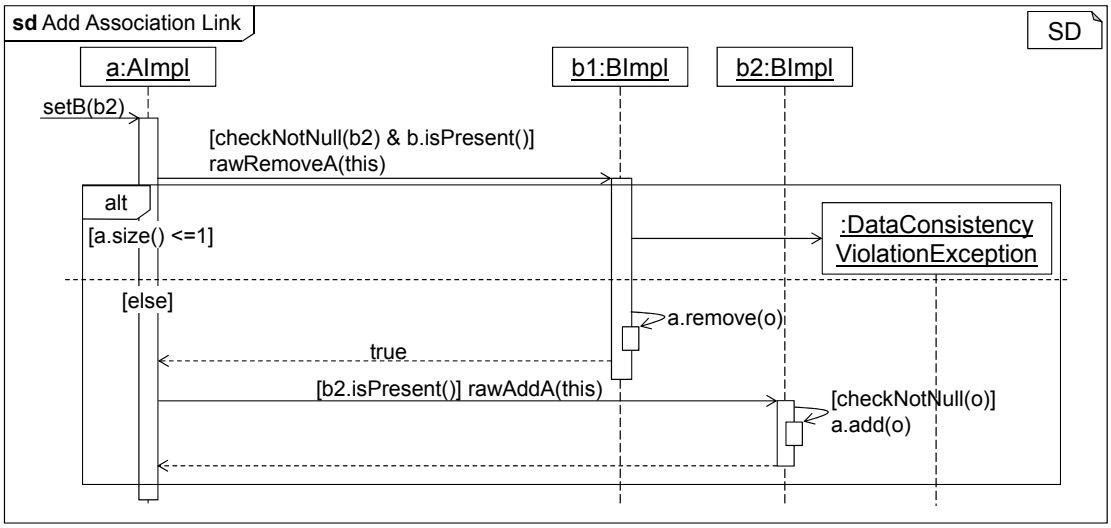


Figure 5.16: An example of handling data consistency when ensuring association consistency. If data consistency is violated, a run-time exception is raised.

The additional checks in this example, e.g., checking if there is at least one association link, are only necessary to ensure data consistency. They are not required for unidirectional associations or for bidirectional associations, where the cardinalities on both sides have a minimum of 0.

The same approach to ensure association consistency and data consistency is used for the `rawSetB()`- and `rawUnsetB()`-methods. However, the `rawUnsetB()`-method implementation throws a `DataConsistencyViolationException`, if the cardinality of the association is changed from `[0..1]` to `[1]`. This is necessary, because if the `rawUnsetB()`-method is called, an already existing association link is to be removed. This will violate data consistency.

5.2.8 Mapping CD4A Ordered Associations

Ordered associations are mapped to the aforementioned methods for associations, as explained in Section 5.2.6, but use the Java `List` type to store the association links to preserve the order in which association links are added and ensure uniqueness via mutators. In addition, the Java source code contains methods for index-based access, adding, and removal as well as iterators for bidirectional link-traversal. These additional methods use the same naming schema as described in Section 5.2.6. Furthermore, all returned subsets are immutable. Note that the «`ordered`»-stereotype for associations with cardinality `[1]` and `[0..1]` has no effect (cf. Section 4.2.4).

An example demonstrating the mapping is shown in Figure 5.17. For presentational reasons, only the additional methods are shown. The ordered association at the top is mapped to additional methods include adding association links at a certain position (ll.2-5 at the bottom) and retrieving a list iterator (ll.7-8 at the bottom), which allows forward and backward traversal but prevents modification by throwing an `UnsupportedOperationException` on modification attempts. Furthermore, positions of elements can be retrieved (ll.9-10 at the bottom) and a sublist, which as well is immutable, (l.11 at the bottom) or only one association link (l.6 at the bottom) from the list of association links can be obtained.

CD4A BankingSystem

```
1 association Deposit -> Share [*] <<ordered>>;
```

...

Java

«GEN»

...

```
1  public interface Deposit {
2                    void addShare(Share o, int index);
3                    void addAllShares(Collection<Share> o,
4                                      int index);
5                    void setShare(Share o, int index);
6                   Share getShare(int index);
7    ListIterator<Share> listIteratorShares();
8    ListIterator<Share> listIteratorShares(int index);
9                     int indexOfShare(Share o);
10                    int lastIndexOfShare(Share o);
11            List<Share> subListShares(int start, int end);
12 }
```

Figure 5.17: For ordered associations additional methods for index-based access, adding, and removal; and iterators for bidirectional link traversal are provided.

Bidirectional ordered associations use the same approach to ensure data consistency as described in Section 5.2.7. This means that additional methods to locally set and remove association links exist.

5.2.9 Mapping CD4A Qualified Associations

Qualified associations denote a strong bond between a key (called *qualifier*), and one or multiple values, i.e., the association links. Each qualified association is mapped to a Java variable with the Java `Map` type, where the qualifier as the key and the association link is the value, to store the association links and additional methods to manipulate the map. If the cardinality is `[*]` and `[1..*]`, then the Java `Multimap` type, which represents a `Map` of a key with a set of values, is used. If the qualified association is ordered, the Java variable's type is `LinkedListMultimap`, which manages lists of association links. Note that ordering is only supported for qualified associations with cardinality `[*]` or `[1..*]` (cf. Section 4.2.5). Table 5.3 summarizes the data types for the different qualified associations by example. The qualified association on the left is mapped to the Java variable on the right. For presentational reasons, the **`association`** keyword for the CD4A associations and the private visibility for the Java variables are omitted.

CD4A Association	**Java variable**
`A [String] -> B [1]`	`Map<String, B> b;`
`A [String] -> B [0..1]`	`Map<String, Optional<B> > b;`
`A [Long] -> B [*]`	`Multimap<Long, B> b;`
`A [Long] -> B [1..*]`	`Multimap<Long, B> b;`
`A [Long] -> B [*] «ordered»`	`LinkedListMultimap<Long, B> b;`

Table 5.3: An overview of the Java variables to store qualified association links. The **`association`** keyword and the private visibility have been omitted.

All Java methods to modify association links of qualified associations use the same naming schema as in Section 5.2.6. These methods depend on the cardinality and existence of the «`ordered`»-stereotype. For qualified association with cardinality `[0..1]`, an example is given in Figure 5.18. The `addB()`-method (l.2 at the bottom) allows to add an association link with a qualifier. It has one parameter for the qualifier (`String key`) and one for the association link (`Optional<B> o`). The type of the second parameter depends on the cardinality of the association, e.g., if the cardinality is `[1]`, the type will be `B`. To check if an association link or a qualifier is contained, the methods in ll.3-4 (at the bottom) exist. Just as for the `addB()`-method, the type of the parameter of the `containsValueB()`-method depends on the cardinality. Furthermore, each stored association link can be retrieved for a particular qualifier (l.5 at the bottom) and the set of qualifiers can be iterated (l.6 at the bottom). An association link can be removed by either removing a qualifier (l.7 at the bottom), in this case all association links of the qualifier are removed, or by removing all qualifiers (l.8 at the bottom). In contrast to ordinary associations (Section 5.2.6), the `sizeBs()`-method returns the number of keys, and the `isEmptyBs()`-method checks if at least one key exists.

```
1 association A [String] -> B [0..1];                                CD4A
```

```
1 public interface A {                                               Java
2              void addB(String key, Optional<B> o);                 «GEN»
3           boolean containsKeyB(String key);                        ...
4           boolean containsValueB(B o);
5       Optional<B> getB(String key);
6  Iterator<String> iteratorKeyBs();
7           boolean removeB(String key);
8              void clearBs();
9               int sizeBs();
10          boolean isEmptyBs();
11 }
```

Figure 5.18: Methods provided for qualified associations with cardinality `[0..1]`.

The mapping shown in the example in Figure 5.18 is also used for qualified associations with cardinality `[1]`. But instead of the `Optional` type, the associations target type is used for parameters and return types. To ensure data consistency for cardinality `[1]`, the Java methods to remove one and remove all association links, i.e., the `clearBs()`- and `removeB()`-method in l.8 and l.7 in Figure 5.18, may throw an exception.

For qualified associations using name qualifiers, which denote an attribute name of the target association end type, the Java method to add an association link, i.e., `addB()`-method in l.2 in Figure 5.18, does not have the qualifier as a parameter, because it can be retrieved from the association link (i.e., the value of the association link's attribute). As a result, qualified associations with a name qualifier and a cardinality of `[0..1]` do not have a key without an association link. If a name qualifier is used, qualifier inconsistencies, i.e., the value of the attribute is changed and requires the key to be updated, may occur when editing existing links. Such inconsistencies are not addressed.

Qualified association with the cardinality `[1..*]` are mapped to additional Java methods to manage the set of association links for each key. Moreover, the method to add an association link has a `put`-prefix instead of an `add`-prefix to denote that an association link is added to the list of association links, as shown in l.2 in Figure 5.19. In addition, because each key stores a set of association links, an iterator (l.7 at the bottom), a contain (l.5 at the bottom), a remove (ll.8-9 at the bottom), and a size (l.10 at the bottom) Java method exist. Note that a method to remove a key does not exist, because it may violate data consistency. For presentational reasons, the example in Figure 5.19 only shows the additional and modified Java methods.

If the cardinality is `[*]` instead of `[1..*]`, there is a Java method to clear and a Java method to check if the set of association links is empty. The name of both additional Java methods is composed as shown in Figure 5.18, i.e., a `clear`- or `isEmpty`-prefix

CD4A

```
association A [String] -> B [1..*];
```

Java «GEN» ...

```
public interface A {
           boolean putB(String key, B o);
           boolean containsKeyB(String key);
           boolean containsValueB(B o);
           boolean containsValueB(String key, B o);
  Iterator<String> iteratorKeyBs();
       Iterator<B> iteratorValueBs(String key);
           boolean removeB(String key, B o);
                     throws DataConsistencyViolationException;
               int sizeBs(String key);
               int sizeBs();
           boolean isEmptyBs();
}
```

Figure 5.19: Methods provided for qualified associations with cardinality `[1..*]`.

concatenated with the association's name. Note that both Java methods are not shown in Figure 5.19, because the clear method would violates data consistency and the set of association links cannot be empty.

Qualified Ordered Associations

A qualified association that is denoted by a «`ordered`»-stereotype represents a qualified association, where the set of association links per key is order-preserving. Note that it does not say anything about the ordering of the keys. In this thesis, it is assumed that the keys are not order-preserving.

Qualified ordered associations are mapped as ordinary qualified associations but with additional Java methods to handle index-based access and handle the order of association links (cf. Section 5.2.8). For example, the mapping shown in Figure 5.20 only shows the additional Java methods, which are the Java methods for ordered associations extended with one additional parameter for index-based access, i.e., `int index`. For presentational reasons all other methods have been omitted. The naming schema follows the naming schema introduced in Section 5.2.6.

Bidirectional Qualified Associations

The difference between bidirectional qualified associations and bidirectional associations is the assumption that the association with the qualifier is responsible for managing the association links. Hence, bidirectional associations with qualifiers on both association ends are not allowed. While this restriction is rather hard, it puts less burden on the

```
1 association A [String] -> B [*] <<ordered>> ;                    CD4A
```

```
1 public interface A {                                             Java
2                void putB(String key, B o, int index)             «GEN»
3                   B getB(int index, String key)                  ...
4    ListIterator<B> listIteratorBs(String key)
5    ListIterator<B> listIteratorBs(int index, String key)
6                 int indexOfB(B o, String index)
7                 int lastIndexOfB(B o, String key)
8            List<B> subListBs(int start, int end, String key)
9 }
```

Figure 5.20: Additional methods for handling qualified ordered associations, which extends the mapping shown in Figure 5.19.

developer than providing methods with qualifiers on both association ends, as presented in [TM05]. As a consequence, the opposite none qualified association end only provides accessors and methods for handling association consistency. If both association ends are qualified, no add or remove methods exist, because the dominating association end, i.e., the association end responsible for managing all instances, has to be defined during the implementation phase by the application developer.

5.2.10 Mapping CD4A Derived Associations

Derived associations can be considered as dynamic, because they do not store any value (cf. [Ste02]). Similar to the mapping in [SBPM09], they are mapped to accessors only as introduced in Section 5.2.6. Each accessor throws a `NotImplementedException`-exception to denote the missing implementation, as described for derived attributes in Section 5.2.5. Furthermore, the additional methods to ensure association consistency do not exist.

For example, the derived unidirectional association as shown in Figure 5.21 at the top is mapped to only one accessor method (`getB()`-method in l.2 at the bottom). Note that the same naming schema is used as described in Section 5.2.6.

```
1 association / A -> B [1];                                        CD4A
```

```
1 public interface A {                                             Java
2   B getB();                                                      «GEN»
3 }                                                                ...
```

Figure 5.21: Derived association (at the top) is mapped to an accessor throwing an exception (l.2 at the bottom) only.

If the CD4A association has the cardinality `[0..1]`, the same mapping applies but with an `Optional` return type for the accessor.

For derived associations with cardinality `[*]` and `[1..*]`, the same mapping to accessors only is used. In addition, Java methods that do not manipulate the association links exist, but they also throw an exception. For instance, the association at the top of Figure 5.22 is mapped to the Java methods at the bottom. Note that the `isEmptyBs()`-method does not exist, if the cardinality is `[1..*]`.

CD4A

```
1 association / A -> B [*];
```

Java «GEN» ...

```
1 public interface A {
2       boolean containsB(B o);
3       boolean containsAllBs(Collection<B> o);
4   Iterator<B> iteratorBs();
5           int sizeBs();
6       boolean isEmptyBs();
7 }
```

Figure 5.22: Mapping derived associations with cardinality `[*]` to access methods only.

This approach of providing accessors only is also applied to derived ordered associations. In this case, the methods shown in Figure 5.22 and the additional accessors for ordered associations (ll.5-10 in Figure 5.17) are provided, each of which throws a `NotImplementedException`-exception. Moreover, qualified derived ordered associations are mapped using the same approach but with additional accessors for qualified associations as described in Section 5.2.9. An example is given in Figure 5.23.

CD4A

```
1 association / A [Long] <-> B [*] <<ordered>>;
```

Java «GEN» ...

```
1 public interface A {
2                 B getB(int index, Long key);
3           boolean containsValueB(Long key, B o);
4           boolean containsValueB(B o);
5   ListIterator<B> listIteratorBs(Long key);
6   ListIterator<B> listIteratorBs(int index, Long key);
7               int indexOfB(B o, Long index);
8               int lastIndexOfB(B o, Long key);
9           List<B> subListBs(int start, int end, Long key);
10   Iterator<Long> iteratorKeyBs();
11              int sizeBs(Long key);
12              int sizeBs();
13 }
```

Figure 5.23: Qualified derived ordered associations are mapped to accessors only.

5.2.11 Mapping CD4A Compositions

A composition is mapped in the same way as associations are (cf. Section 5.2.6), i.e., a variable to store the link with accessor and mutator methods. This corresponds to the weak interpretation of composition (cf. [Rum12, Lan16]), which does not ensure that a new part object of the composition is created. However, this mapping does not regard the lifetime dependency of the whole and its parts. Such concerns are handled in Section 7.2.2. Furthermore, this understanding of a composition does not fully cover the required semantics of composition, as it is discussed in [Ges08].

The example in Figure 5.24 shows the mapping for an unidirectional composition (at the top). As shown in l.2 (at the bottom), the variable `b` stores the association link. It is set in the mutator (ll.10-13 at the bottom). In addition, an accessor exists (ll.15-17 at the bottom). All Java methods use the name schema described in Section 5.2.6.

CD4A

```
1 composition A -> B [1];
```

Java «GEN» ...

```
1  public class AImpl implements A {
2    private B b;
3
4    protected AImpl(String aaaa, B b) {
5      super();
6      checkNotNull(b);
7      setB(b);
8    }
9
10   public void setB(B o) {
11     checkNotNull(o);
12     b = o;
13   }
14
15   public B getB() {
16     return b;
17   }
18 }
```

Figure 5.24: An example of mapping CD4A composition to Java source code.

An alternative understanding of the composition is presented in [Rum12]. It assumes the composition to be frozen and have the cardinality `[1]` (from whole to part). Moreover, [AHMM07, SBPM09] present an understanding of composition, which is based on additional containers. Each container is a superclass of the whole and the part that stores composition related information. This understanding still has drawbacks, e.g., composition life time (cf. [Ges08]).

5.3 Method for Handling Mandatory-to-Mandatory Associations

A restriction of the CD4A-to-Java mapping is that even if syntactically correct source code is produced, the resulting Java source code implementing the data structure may not be instantiated (cf. "*chicken-and-egg issue*" [BFL13]). For example, consider the example in Figure 5.25. In this example, the CD4A model at the top is mapped to the implementing Java source code on the bottom left and bottom right. Because both association ends define a mandatory association, it is not possible to instantiate any of the classes. The reason is shown in the constructors of both classes ll.3-6 at bottom left and bottom right.

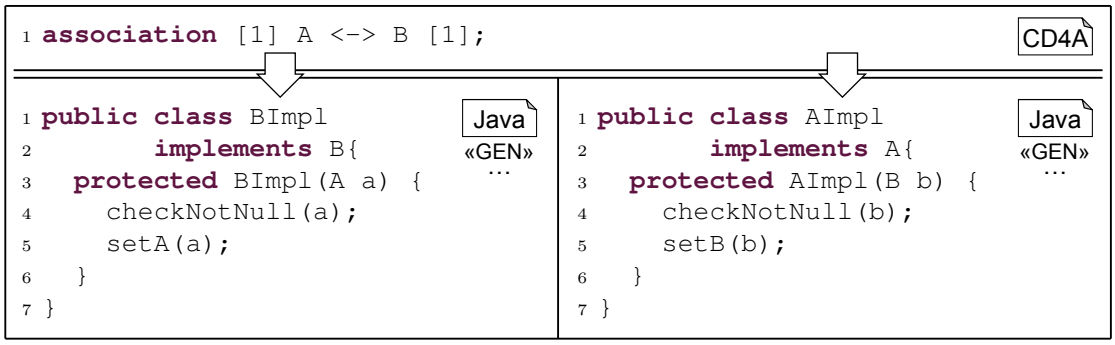


Figure 5.25: The association at the top is regarded in the mapped Java class `BImpl` (bottom left) and the Java class `AImpl` (bottom right).

One possible realization is to extend the mapping of each association source with an additional constructor with the parameters of the association target's type such that the object is created and set within the constructor, as described in [BFL13]. However, this mapping does not scale for transitive mandatory associations. Another drawback is that naming conflicts in the resulting constructor's parameter list may occur. For example, when there are multiple mandatory associations to the same type.

Another technical realization considers (i) only partial inconsistency of the data structure, where it is necessary, and (ii) avoids introducing a wrapper for the instance itself. The primer is necessary to restrict the modifications of the object to avoid further inconsistency. The latter is required to avoid cloning instances, which may be time consuming. Technically, it can be founded on the *Prototype*-Pattern [GHJV95]. Each object is allowed to have a prototype state, in which partial inconsistency is allowed only, e.g., for a particular association. In this state, the instance itself is regarded as prototypical and every change -other than making the instance consistent- leads to an exception.

For instance, the example from Figure 5.25 is realized as shown in Listing 5.2. First, a private default constructor is added such that it cannot be used in subclasses or

accessed from outside (ll.10-12). In general, this constructor has to contain all other attribute and mandatory associations, if any are present. It is called from the static `createPrototype()`-method (ll.14-16) to retrieve an instance, which is partially inconsistent, because the association has not been set yet.

When creating a prototype of the `BImpl` class, the only method allowed to be called is the `setA()`-method, because this method has to be called to make the instance consistent. All other methods check whether the instance is still a prototype and throw an exception if this is the case, e.g., l.19, and l.25. Note that the additional methods for handling association consistency do not require this check.

To leave the prototype stage and create the valid object, the `leavePrototype()`-method has to be called (ll.18-22). This method can only be called during the prototyping stage. It ensures that the instance is consistent, i.e., in this case the association is set (l.15). If the prototype stage has been left, it cannot be entered again.

Such an approach can be encapsulated in, e.g., a Builder-Pattern (cf. Section 7.2.1). Moreover, this approach can be realized by either manually-written extensions (as described in Chapter 6) or it can be synthesized by default when such situations are detected in the model.

Java

```
1 public class BImpl implements B {
2   private boolean $prototype = false;
3   private A a;
4
5   protected BImpl(A a) {
6     checkNotNull(a);
7     setA(A);
8   }
9
10  private BImpl() {
11    this.$prototype = true;
12  }
13
14  static BImpl createPrototype() {
15    return new BImpl();
16  }
17
18  void leavePrototype() {
19    checkArguments(this.$prototype);
20    checkNotNull(a);
21    this.$prototype = false;
22  }
23
24  public A getA() {
25    checkArguments(!this.$prototype);
26    return this.a;
27  }
28
29  public void setA(A o) {
30    checkNotNull(o);
31    if (this.a != null) {
32      this.a.rawUnsetB();
33    }
34    o.rawSetB(this);
35    this.a = o;
36  }
37 }
```

Listing 5.2: An implementation example of the CD4A model in Figure 5.25 to implement CD4A composition.

Chapter 6

Generated Code Customization via Handcoded Extensions and Hot Spots

In Chapter 4, the CD4A ML to describe the structured data managed by data-centric applications has been introduced. Afterwards, a CD4A-to-Java source code mapping, which describes how to systematically derive a consistency-ensuring data structure, has been described in Chapter 5. However, defaults are used by the mapping to resolve semantic variation points (cf. *RE-5-2*) and to regard underspecified implementation details such as derived attributes and derived associations. Therefore, the data structure has to be manually extended to customize defaults or to provide missing implementation details (cf. *RE-1-2*).

In general, extending a ML with implementation concerns has shown reduced acceptance among application developers (cf. [KBR11]). Hence, mechanisms to directly customize generated source code have been proposed [GHK+15a, GHK+15b]. In agile development of data-centric applications, as presented in Section 3.3, a mechanism that enables manually-written customizations of generated source code has to consider the following properties:

- Ensure that manually-written source code is used instead of generated source code.
- Support iterative and incremental software development processes by ensuring that manually-written source code is not overridden by MDD tools.
- Strictly separate generated and manually-written code on artifact-level, because generated source code artifacts are regarded as disposable products that should not be versioned, managed, or inspected [SVC06]. However, handcrafted artifacts should not be modified by a generator. Generated artifacts should also not be modified manually.
- Data consistency is still ensured (as much as possible), although violations cannot be prevented by design in all cases.

The goal of this chapter is to introduce the Extended Generation Gap-Pattern, which is a design pattern that regards the aforementioned considerations and has been developed in the course of this thesis.

This chapter is structured as follows. First, general considerations of manually customizing generated source code are discussed in Section 6.1. Afterwards, the Extended Generation Gap-Pattern is introduced in Section 6.2. This approach is extended with the benefits of predefined hot spot customization, as explained in Section 6.3. Finally, a method for its effective use is described in Section 6.4.

6.1 General Considerations of Handcoded Extensions

Each approach to support customizations of the generated source code has to be explicitly integrated into the overall generated software system's architecture, i.e., it is explicitly designed to regard handcoded extensions (cf. [SVC06]). For example, by assigning appropriate visibility to methods, i.e., `public` or `protected`, or using design patterns [GHJV95]. Depending on the handcoded integration approach, a deep level of understanding of the generated source code might be required. For example, a code-frame-based approach, i.e., the code generator produces marked code frames in artifacts, requires detailed knowledge, because manually-written source code is placed directly in the generated one.

In this thesis, we aim for an approach to integrate *handcoded extensions* that suits the targeted architecture (cf. Section 2.3.1) and reduces the required knowledge by relying on interfaces of the generated source code instead. Handcoded extensions are understood as follows:

Definition 14 (Handcoded Extension)**.** *A handcoded extension is a manually-written source code artifact created to adapt or extend one generated source code artifact. It is written in the same GPL as the generated source code.*

Handcoded extensions can be seen as an approach to provide a familiar mechanism for application developers (cf. Section 3.1.3), because existing programming languages and tool support are reused. However, a separation of generated and non-generated artifacts, which is implicitly assumed in Def. 14, creates a significant implication for the generated software system's architecture and for the handcoded integration approach, which are presented in the following as general demands for a customizable architecture of the generated software system.

6.1.1 Separation of Generated and Non-Generated Artifacts

Direct modifications of generated source code hold a plethora of challenges, e.g., consistency between model and generated source code, and build management [SVC06]. Although, *round-trip engineering* [BGSZ08] attempts to reduce these drawbacks, automated conversion cannot provide the required level of abstraction (cf. [Sel03]) and is only applicable for design models (cf. Section 4.1).

From a technical point of view, current tools and code generators can manage intermixed generated and non-generated source code (cf. [SVC06]). Such sophisticated tools process the intermixed source code to respect any form of *evolution* of the input model [RGLR13, GLRR13, GLRR15] and protect manually-written source code. However, this approach conflicts with the objective of MDD, because generated source code is handled as a primary development artifact when developing and deploying the overall software system. Furthermore, the input model may not reflect all aspects of the generated software system (cf. [HBR00]).

RE-1-2-1 Separation of Generated and Non-Generated Artifacts: In contrast to intermixed generated and non-generated code, a clear separation of both has been proposed [HBR00, SVC06, FBHK+07, GHK+15a, GHK+15b], which is also followed in approach used in this thesis. Such a separation has the following advantages:

(i) Reduced complexity of code generators that is introduced by processing and preserving non-generated code (cf. [SVC06]).

(ii) It allows to version handcoded extensions independently.

(iii) It enables repeatable generation.

(iv) In the case where object-oriented target languages are used, additional in-depth knowledge of the implementation is not required, because knowledge of the generated interfaces is sufficient.

A separation of generated and non-generated source code is essential for repeatable code generation, as targeted in this thesis (cf. *RE-3*). It ensures that generated source code is not regarded by application developers but handwritten code can be injected through various plug-in, delegation, and inheritance mechanisms [SVC06] even if the model evolves. In contrast, in a one-shot code generation approach, source code is generated only once and manipulated by application developers such that handwritten code and generated source code is intermixed. In a one-shot code generation approach, the model does not adequately reflect evolution of the generated source code. Which is not the case in repeatable code generation, where the model is considered as the primary development artifact.

A generated software system's architecture supporting such separation of artifacts can be realized through the use of design patterns [GHJV95] and explicit spots, which allow for predefined customizations [Pre95]. Note that these guidelines rely on object-oriented principles and, hence, may not provide a sufficient solution for other target languages.

6.1.2 Override-Static-Pattern

Besides a clear separation of artifacts into generated and handcoded artifacts, a major concern is to ensure that manually-written code is actually used throughout the entire

generated and non-generated source code (cf. Chapter 6). In other words, if a surrogate, which is a handcoded extension, of a generated class exists, then only the surrogate is instantiated but not the generated class.

An implementation pattern to realize such an approach is the *Override-Static*-Pattern, which is a static variant of the Delegator-Pattern [GHJV95]. It suits these requirements better than patterns for object creation such as the *Factory* or the *Builder*-Pattern [GHJV95], because the object itself defines what instance is created. Note that in the remainder of this section, the generated and non-generated source code is assumed to be Java. Hence, the pattern is shown for Java, but is applicable for other object-oriented programming languages as well.

The Override-Static-Pattern's main elements are two static methods to define the instance that is to be created. To demonstrate this pattern, we use the example in Listing 6.1. An instance of the `A` class is created by calling the `get()`-method (ll.5-8). It delegates the call to the `init()`-method (ll.10-14), which defines the instance that is created, and stores the created object in the `a` static variable (l.12). Afterwards, this newly created instance is returned (l.7). Since the constructor is protected (ll.16-17) objects can only be created by calling the `get()`-method.

Java

```
1 public class A {
2   protected static A a = null;
3
4   // object creation
5   public static A get() {
6     A.init();
7     return a;
8   }
9
10  public static void init() {
11    if(a == null){
12      a = new A();
13    }
14  }
15
16  protected A () {
17  }
18
19  // delegation
20  public static void action() {
21    get().doAction();
22  }
23
24  protected void doAction() { ... }
25 }
```

Listing 6.1: The Override-Static-Pattern implementation to instantiate objects.

To integrate handcoded extensions in the this pattern, the manually-written code has to be an extension of the generated artifact, i.e., the manually-written code uses the `extends` construct for Java classes to extend the generated source code, and implements a static `init()`-method, which instantiates the handcoded source code (in this case a Java object has to be instantiated). An example is shown in Listing 6.2.

```
1 public class AExt extends A {                                    Java
2   public static A init() {
3     a = new AExt();
4   }
5
6   protected void doAction() {
7     ...
8   }
9 }
```

Listing 6.2: A handcoded Java class for the generated `A` class shown in Listing 6.1.

When the Override-Static-Pattern is used in the generated source code, an integration approach for handcoded extensions has to adapt l.6 in Listing 6.1 to reference the manually-written `init()`-method. In this example, l.6 in Listing 6.1 has to be changed to `AExt.init()`. Moreover, to ensure that generated source code cannot be instantiated anymore, the generated class has to become `abstract` and whenever the generated `init()`-method (ll.10-14 in Listing 6.1) is called, an exception has to be thrown. The `get()`- and `init()`-method of the Override-Static-Pattern ensure that the interface for object creation does not change, i.e., the `get()`-method can still be used, but the object created is different.

In addition to object creation, the Override-Static-Pattern facilitates overriding static methods to ensure that the provided interface remains the same but the implementation changes. For example, the static `action()`-method in ll.20-22 in Listing 6.1 delegates each method call to the `doAction()`-method in l.24 using the `get()`-method (l.21). This delegation allows to override the implementation of the `doAction()`-method in a subclass (e.g., ll.6-8 in Listing 6.2), which is then called due to the `get()`-method call. As a result, the implementation of a static method can be overridden.

6.2 Integration of Generated and Non-Generated Code

An integration approach for generated and non-generated object-oriented source code has been developed in our previous work [GHK+15a, GHK+15b]. It is based on the *Generation Gap*-Pattern, which is an approach to extend the generated code with manually-written code by reusing object-oriented inheritance [Vli98, SVC06, Fow10]. However,

the Generation Gap-Pattern assumes that for each generated file a manually-written file exists (even if manually-written code is not necessary). Furthermore, it does not provide means to extend the generated interface.

Our proposed *Extended Generation Gap*-Pattern extends the Generation Gap-Pattern to provide a framework independent approach that relies on general object-oriented programming concepts and tackles the disadvantages of the approach it is based on. It has the same prerequisites as the Generation Gap-Pattern (cf. [GHK+15a, GHK+15b]) but rather than demanding in-depth knowledge of the generated source code, it only requires knowledge of the generated interfaces to add handcoded extensions.

To demonstrate the pattern, we use the example in Figure 6.1. Assuming that the `Transaction` interface and the `TransactionImpl` class in Figure 6.1 are generated (denoted by the «GEN» tag), a naming convention for the handcoded extensions (denoted by the «HC» tag) is introduced. Each manually-written implementation extension, which extends the generated implementation, is specified by the "`EIMP`" suffix, e.g., `TransactionEIMP` implementation.

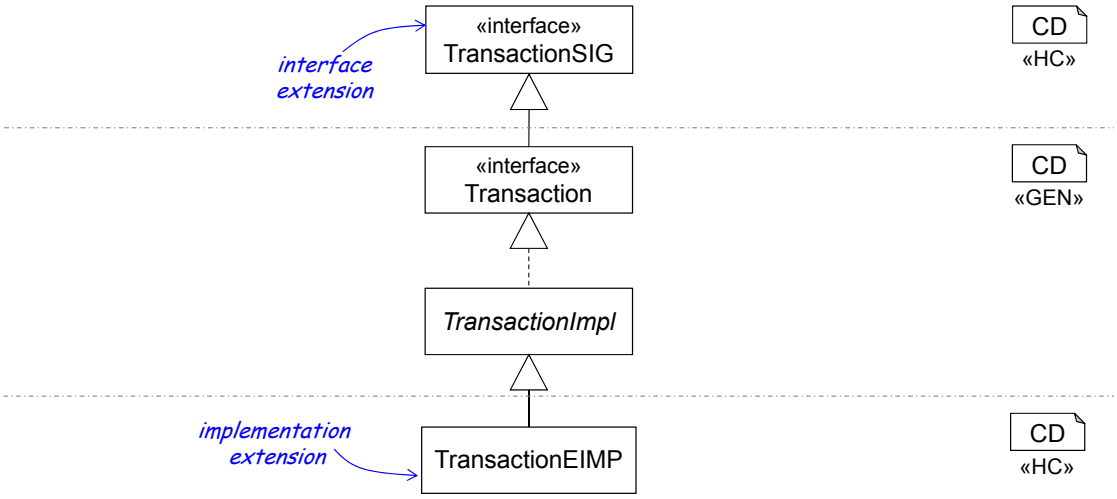


Figure 6.1: The Extended Generation Gap-Pattern allows for implementation extensions (`TransactionEIMP`) and interface extensions (`TransactionSIG`).

Although implementation extensions enable customization of the generated source code, customizations such as new methods cannot be accessed from other classes without explicit casting. In addition, because CD4A models do not contain technical classes, i.e, classes required by external frameworks to make use of particular functionality, interface extensions are provided. A manually-written interface extension, which allows to add additional methods to the generated interface, is denoted by the "`SIG`" suffix, e.g., `TransactionSIG` interface in Figure 6.1. Such a naming convention emphasizes the

clear separation of generated and non-generated code (cf. *RE-1-2-1*). Technically, the generated interface extends the manually-written interface extension, to enable use of the added methods through the generated interface.

Figure 6.1 also shows the influence of the proposed approach on the generated systems' architecture, which is introduced in Figure 7.1 on page 110. Namely, interface extensions require the generated interface to extend the manually-written interface, as shown by the `TransactionSIG` interface.

Technically, to realize the implementation extensions, the Override-Static-Pattern (cf. Section 6.1.2) can be used. For example, for the example in Figure 6.1, the `TransactionImpl` provides a static `get()`-method and a static `init()`-method, which is implemented in the manually-written `TransactionEIMP` class. Moreover, a tool realizing this approach, e.g., a code generator, has to automatically detect handcoded extensions and adapt the generated source code to ensure that manually-written source code is regarded. As a result, regeneration is required whenever a new handcoded extension is added, because the generator needs to adapt the generated code. Regeneration is also required when the input model evolves. However, tool support may conflict with the absence of the generated code while development of handcoded code, because, e.g., the code generator has not been executed yet. Hence, every kind of reporting is crucial for debugging (cf. *GR-3*).

In general, manually-written implementation and interface extensions can be created for each resulting class of the mapping introduced in Chapter 5. However, from a technical point of view, interface extensions may require an implementation extension. In the following section, implementation approaches to overcome this restriction are discussed.

6.2.1 Implementation of Interface Extensions using Java-Default Interfaces

A technical realization of the Extended Generation Gap-Pattern requires for each interface extension an implementation extension, as shown in Figure 6.1. In this example, the `TransactionSIG` interface extension requires the `TransactionEIMP` implementation extension. Otherwise, the generated source code may not be syntactically valid, because it does not implemented the additional methods introduced in the interface extension. This approach has been chosen in this thesis, because it only requires basic object-oriented concepts [Eli94].

To avoid this restriction, additional target language constructs such as default implementations, which have been introduced by version 8 of the Java programming language, can be used. Following this, the Extended Generation Gap-Pattern can be realized as shown in Figure 6.2. In this example, the `TransactionSIG` interface extension uses Java's default implementation to implement added methods. However, in this case the handwritten signature extension has to inherit from the generated interface to use the generated accessors and mutators. Moreover, the generated implementation (`TransactionImpl` class) has to implement the manually-written interface extension.

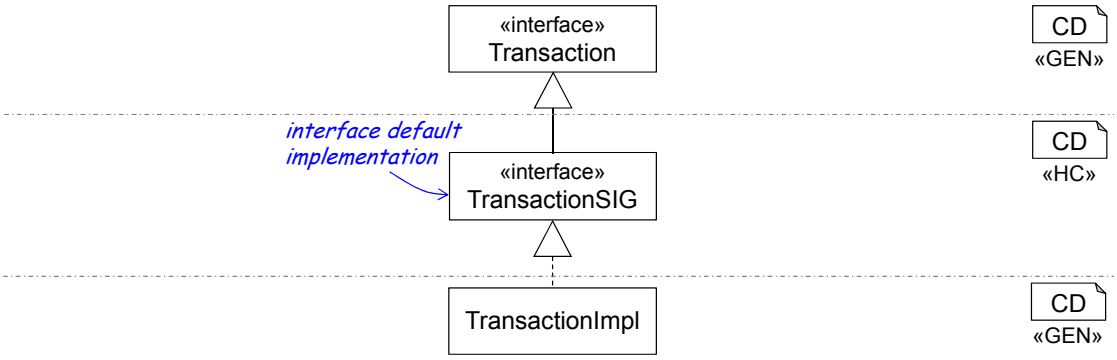


Figure 6.2: An example of implementing interface extensions using Java default methods.

While this approach avoids the need to provide an implementation extension, it does not make manually-written interface extensions explicit in the generated interface. In other words, for the example in Figure 6.2, accessing the methods added by the manually-written `TransactionSIG` interface, a cast from the generated `Transaction` interface to the handcoded one is required.

6.2.2 CD4A Hierarchy and Handcoded Extensions

For the proposed mapping in Chapter 5, a general guideline is to preserve hierarchy defined in the CD4A model, as it is proposed in [Rum12]. This principle is also applied to the Extended Generation Gap-Pattern, when extending the generated source code with handcoded extensions.

To demonstrate how this principle is applied, the CD4A model shown in Listing 6.3 is used. It presents the `CheckingAccount` class, which is a subclass of abstract `Account` class. According to the mapping in Chapter 5, for each CD4A class an interface and an implementation is generated. Note that this listing is a simplified excerpt of the UML CD shown in Figure 4.1 on p. 48.

```
1 abstract class Account { ... }                    CD4A BankingSystem
2                                                          ...
3 class CheckingAccount extends Account { ... }
```

Listing 6.3: An excerpt of the CD4A model in Figure 4.1 with a hierarchy of classes.

When adding interface and implementation extensions to the generated classes using the Extended Generation Gap-Pattern, the generated source code respects the hierarchy of the `CheckingAccount` class by adding the inheritance relation (1) in Figure 6.3. However, the implementation extensions `AccountEIMP` is not available in its subclass, because the `CheckingAccountImpl` class has to extend the manually-written

`AccountEIMP` class. Hence, this inheritance relation (②) in Figure 6.3) has to be added automatically by the MDD tool (i.e., generator) (cf. [HBR00]).

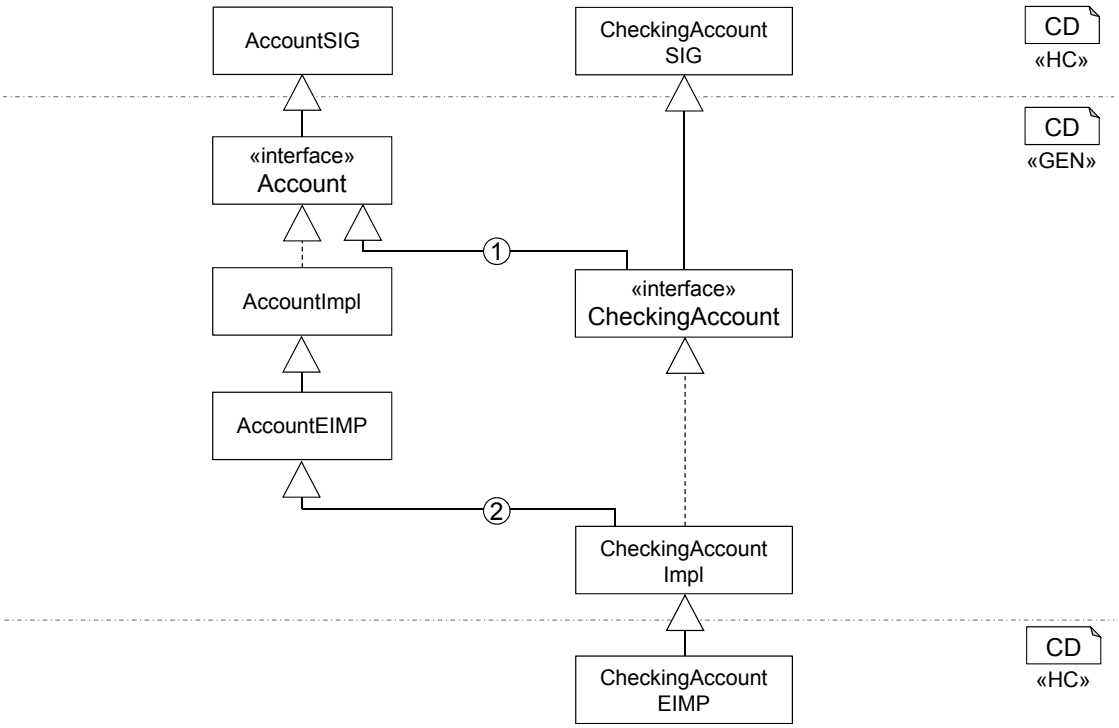


Figure 6.3: An example showing that adding handwritten extensions to subclasses requires adaptation of the generated source code.

A result of such code generator behavior is that adaptations have to be designed a priori by generator developers to enable full support of the Extended Generation Gap-Pattern. Nevertheless, this is in general a necessity for adaptations (cf. [ABKS13]).

6.3 Customization via Hot Spots in Generated Source Code

Pree defines *hot spots* as "those aspects that have to be kept flexible" [Pre95]. A hot spot introduces certain variability into predefined design solutions and is often realized using design patterns [SLK06]. This "plugged-in" application specific implementation allows to create an individual design solution by adding customizations to the generated source code while regarding it as a black-box [Sch97] (cf. *PR-4*).

Frameworks and GUIs heavily rely on hot spots to provide a generic solution that can be adapted to particular needs. Hence, hot spots are understood as follows:

Definition 15 (Hot Spot)**.** *A hot spot is a dedicated spot of predefined customization in the generated source code that is a priori planned for handcoded extensions.*

In general, hot spots are independent of external frameworks, because they are realized using target language concepts and, hence, are part of the framework provided by the generated software system. Technically, hot spots are *hook methods*, which are place holder methods called by *template methods*, which are methods defining abstract behavior or generic flow of control [SLK06]. Such hook and template methods can be identified using the *hot-spot-driven development process* [Pre00].

Typically, hot spots are used in one of the following ways:

- Adaptation: An existing default method (often having an empty method body) is adapted by means of overriding it to provide a method implementation.
- Delegation: An implementation for an interface is injected.

Using hot spots to adapt generated source code harbors the potential to corrupt the generated code. Hence, measures need to be taken to avoid such pitfalls. Besides producing corrupt source code, it has to be ensured that hot spots are used such that application developers experience a benefit. In the worst case scenario (if hot spots are not documented), they are useless. Therefore, in this thesis, a documentation specifying the hot spot's location and purpose to ensure its use and acceptance by application developers is proposed. Such a hot spot documentation has to be deployed with the generated source code and the code generator, if the generated software system is planned to be extended. In this thesis, the following design guidelines for hot spot documentation are defined:

1. Use a unique name and location for the hot spot.
2. Only document hook methods with their purpose for explicit customization, because template methods require in-depth knowledge of the generated architecture.
3. Document connections between hot spots, e.g., if hot spots influence each other.
4. Separate hot spots into optional (hot spots that can be used for customization) and mandatory (hot spots that have to be used in order to make the overall software system work).

6.4 Methods for using Handcoded Extensions

An integration approach for generated and non-generated code is always accompanied by a consistent method to, e.g., name artifacts in a particular way or edit predefined

artifacts. Such a method is not allowed to change, for all handcoded extensions, because otherwise the preceding handcoded extensions have to de adapted. In addition, such a methodology requires clear instructions on how to use the proposed integration approach. Hence, a methodology for the Extended Generation Gap-Pattern to add handcoded extensions is illustrated in the Figure 6.4.

The method presented provides an abstract overview. The technical realization of hot spots and the code generator are presented in Chapter 9. For this implementation all available hot spots are documented in Appendix F.

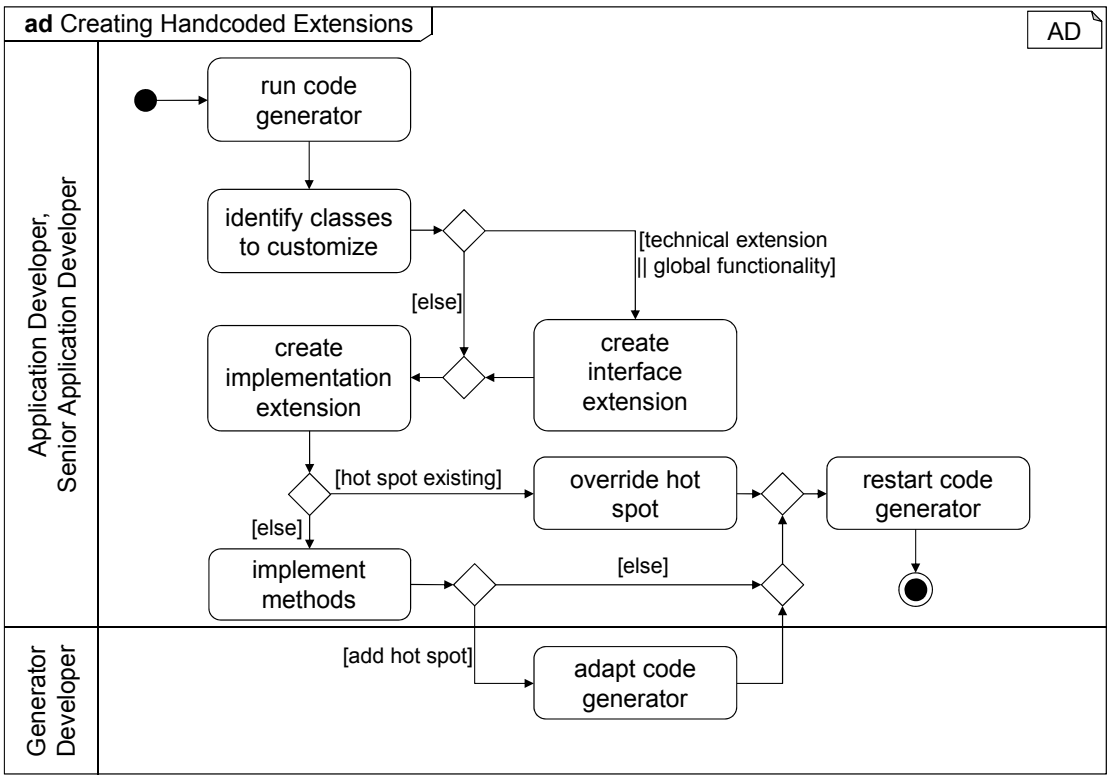


Figure 6.4: A method for using interface and implementation extensions of the Extended Generation Gap-Pattern.

Before the source code can be adapted, the (senior) application developer has to execute the code generator and identify the generated class that has to be manually-extended. This method implies that if the customization affects multiple classes, it has to be broken down to one class to consider. Afterwards, if the extension is (i) motivated by technical demands, i.e., to conform to a particular interface provided by a framework, or (ii) if the extension has to be globally available, e.g., a method that is accessed from different classes, then an interface extension has to be created.

In contrast, if customizations (i) require adaptation or extension of the generated implementation or (ii) an interface extension for the generated artifact has been previously created, then an implementation extension has to be created by either the (senior) application developer. Note that if another approach to realize the handwritten code integration is used, as discussed in Section 6.2.1, the implementation extension might not be necessary for an interface extension.

When realizing an implementation extension, the available hot spots are checked to identify if existing hot spots can be reused. If no hot spot is defined, the generator developer can adapt the code generator in order to provide such a hot spot. This is helpful when reuse is regarded and the code generator is used in several software projects.

Finally, if the manually-written class has been created after the code generator execution, the code generator has to be restarted, because it detects the newly created handcoded extension and adapts the generated source code, as described in Section 6.2.

6.4.1 Extending and Associating External Data Types in CD4A Models

Handcoded extensions can also be used to remove the limitation of CD4A, when using external data types, as explained in Section 4.2.5, i.e., (a) inheriting from an external data type and (b) creating bidirectional associations to external data types is not supported. Note that the primer, i.e., (a), is only a limitation of the chosen technical realization. It can be removed by using appropriate parsers to process the external data types.

To demonstrate how these limitations can be removed, the example in Figure 6.5 is given. Assuming the `Customer` class is modeled and has to (a) extend the external class `Person` and (b) associate the external data type `Address`. With the previously mentioned restrictions, this is forbidden by the CD4A ML, because when using this model in a MDD approach, the generator has to parse the external data type to detect the correct constructor that has to be implemented or has to adapt an external data type in order to add the required language constructs to manage the bidirectional association.

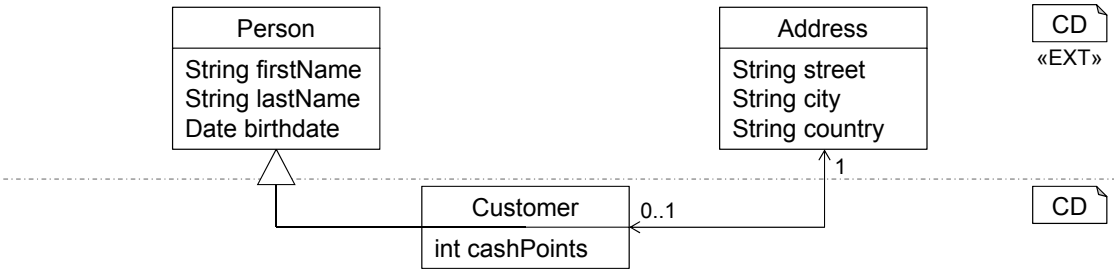


Figure 6.5: A UML CD showing of a `Customer` referencing externally defined classes.

The first restriction, i.e., (a), can be overcome by employing the Delegation-Pattern [GHJV95]. In particular, an additional class is modeled, which references the external

class and delegates the method calls of accessors and mutators to the external class. In the example in Figure 6.6, the delegating class is `ExternalPerson`. This class serves as a wrapper for the external class, which is implemented using handcoded extensions. Nevertheless, a disadvantage of this approach is that protected variables of the external superclass cannot be accessed.

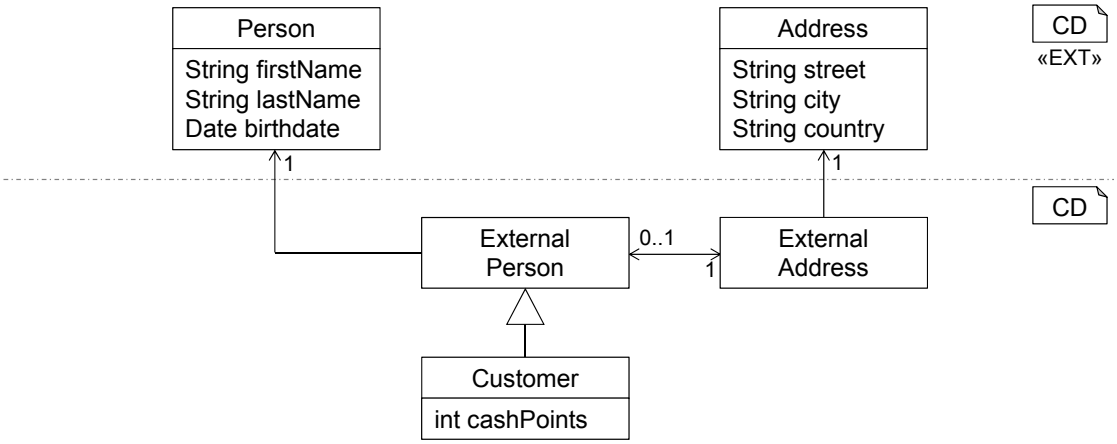


Figure 6.6: A UML CD showing the proposed use of externally defined classes for the example in Figure 6.5.

The second restriction, i.e., (b), can be resolved similarly by introducing a Delegator-Pattern as well. For example, the `ExternalAddress` class on the right-hand-side in Figure 6.5 is the delegator for the external data type. The introduced delegator handles the association and delegates calls to the external data type.

Chapter 7

A Customizable Data-Centric Infrastructure

A data structure resulting from a CD4A-to-Java source code mapping, which has been described in Chapter 5, forms only the core of a data-centric application. Such a data structure has to be extended with a data-centric infrastructure (cf. Section 2.3) to implement a data-centric application, which facilitates persistent management of the data structure via a GUI (cf. *RE-1*).

A data-centric infrastructure that efficiently and successfully supports MDP and MDD of data-centric applications, as described in Section 3.3, and supports MDD tool reuse, has to address:

- Modularity, which describes the property of the data-centric infrastructure to be used as a whole or in parts. This way a data-centric application supports standalone and framework-like use, i.e., parts of a data-centric application are used in a manually-written software system.
- Customizability, which describes the property of the data-centric application to be extended and customized. As a result, varying requirements in the development of data-centric applications and data-centric application prototypes can be realized.

The data-centric infrastructure developed in this thesis uses a layered architectural style to segregate the overall complexity into dedicated layers (cf. Section 2.3.1). It is shown in Figure 7.1. Each layer consists of a model-independent part (`RTE` component) and a model-dependent part (`generated` component). The model-independent part is provided by a RTE, which is deployed with each data-centric application and contains non-generated source code required to compile and run the generated source code. The model-dependent part is mapped from a CD4A model describing the managed data. In addition, each layer may consist of a `manually-written` component, which represents handcoded extensions of a layer, as explained in Chapter 6; and a `standard components` component, which represent the standard libraries of the GPL or additional external libraries.

The dashed lines in Figure 7.1 indicate the uses-relation between components and follow the general guideline for layered architectures (upper layer is allowed to use the lower layer). However, (1) in Figure 7.1 violates this general rule, because the persistence

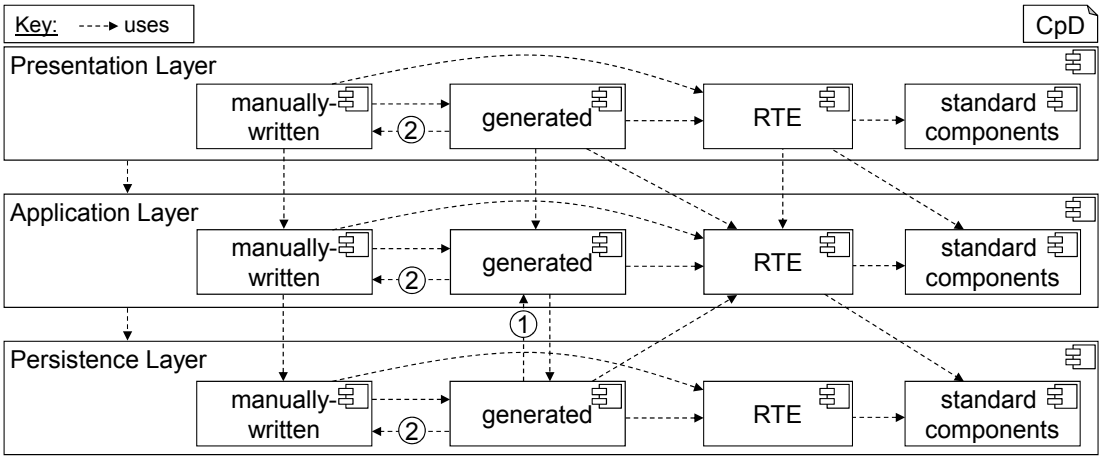


Figure 7.1: Overview of the developed layered architecture for data-centric applications.

layer instantiates the data structure objects. Another unintuitive static relation is (2), which results from the Extended Generation Gap-Pattern that uses generated source code to add manually-written source code (cf. Chapter 6).

This layered architectural style separates data-centric applications from general InfoSyss, which use a 3-tier architecture [BMR+96], where a tier represents a physically structuring approach for the overall infrastructure. In contrast, a layer represents a structuring approach of a software system. However, to persistently manage the data structure, a model-independent persistence infrastructure is used in this thesis.

The goal of this chapter is to describe a method to realize a data-centric infrastructure for data-centric applications and explain the main aspects of the technical realization. Hence, this chapter is structured as follows. First, general considerations and architectural drivers for a data-centric infrastructure are discussed in Section 7.1. Afterwards, systematic mappings of the CD4A ML to model-dependent parts in the application layer (cf. Section 7.2) and presentation layer (cf. Section 7.3) are presented. Next, the main aspects of persistence infrastructure's technical realization are described in Section 7.4, and the mapping for the persistence layer (cf. Section 7.5) is explained. Finally, a method for consistent data migration is presented Section 7.6.

7.1 General Considerations and Architectural Design Drivers

From the definition of a data-centric application (cf. Section 2.3), the functionality of a data-centric infrastructure to manage a data structure can be partitioned into: (a) operations allowed on the data structure, (b) a GUI to execute these operations and display the data structure (cf. *PR-2*), and (c) a connection to a persistence infrastructure

to store created instances of the data structure (cf. *RE-2*). Each concern is explained in the remainder of this section.

(S)CRUD Functionality. Each data-centric application provides functionality to manipulate the data structure. Such functionality is separated into *Create, Read, Update*, and *Delete* (CRUD) operations. In addition, this basic functionality is extended with Search operation (S).

While the CRU operations are provided by the constructors, accessors, and mutators in the proposed mapping (cf. Chapter 5), the D operation has to be handled by the data-centric infrastructure, because deletion may violate data consistency. Due to the design and defaults chosen for the CD4A-to-Java mapping, the generated data structure cannot handle data consistency on its own. For example, an unidirectional association with a mandatory `[1]` cardinality will violate data consistency, if the referenced object is deleted.

Graphical User Interface (GUI). A GUI uses the provided (S)CRUD operations and offers a structured, consistent, and pleasing view of the data. Consistent in this case means that for the same managed element different views show the same content. Pleasing is, however, very subjective and has to be defined in each particular case. Hence, in this thesis, a default mapping for CD4A to GUI is provided. Adaptations of the GUI can be made via handcoded extensions of the data-centric infrastructure.

Commonly, a GUI for data-centric applications provides two types of views (cf. Section 2.4). One view for listing all elements of a certain type (*ListView*) and another view for editing model elements (*EditView*).

Persistence Each data-centric application is only complete, if managed data can be stored in a persistent way. For lightweight client applications, as understood in Section 2.3, this implies that a persistence infrastructure is used, where data is stored in a database. In this thesis, it is assumed that the persistence infrastructure is unaware of the CD4A model (cf. *RE-2-2*). As a result, server-side data consistency concerns are not possible, as further explained in this section.

In addition, when one persistence infrastructure manages multiple data structures manipulated by multiple data-centric applications, it has to provide role-based access control to ensure that each data-centric application has only access to its managed data (cf. *RE-2-1*).

The technical realization of a data-centric infrastructure fulfilling the aforementioned considerations is also influenced by customization (cf. *RE-1-2*), type-safety (cf. Section 5.1), and modularity concerns (cf. *GR-1*), as it is targeted in this thesis. Hence, in the remainder, the architectural impact to regard these considerations is presented.

7.1.1 Architectural Impact of Infrastructure Customization

There are many notions to describe the property of a software system to allow for changes to fit particular requirements such as “adaptable”, “customizable”, “tailorable”, and “malleable”(cf. [SQK06]). We consider *infrastructure customizability* as follows:

Definition 16 (Infrastructure Customizability)**.** *Infrastructure customizability is the property of the provided infrastructure to be adapted to a particular requirement by means of handcoded extensions.*

Technically, this approach is realized using the Extended Generation Gap-Pattern (cf. Section 6.2). Its architectural impact on the data-centric infrastructure is visible by the use of the Override-Static-Pattern (cf. Section 6.1.2), which is applied to every generated Java class to create objects and facilitate handcoded extensions. Hence, in the remainder of this section, this pattern is omitted in the description of the data-centric infrastructure for presentational reasons.

7.1.2 Type-specific Method Invocation via Double Dispatching

A data-centric infrastructure has to enforce static type-safety to avoid unpredictable application behavior (cf. Section 5.1) because of reflective access to ensure type-safety of the managed data structure (cf. Section 5.1). A design pattern to ensure static type-safety while reconstructing type information is the *Double Dispatching*-Pattern, which is an approach to dynamically select a method based on the run-time type of its parameter [BCV05]. This design pattern prevents use of reflection or casts and thus allows to enforces static type-safety. This pattern is essential for the developed data-centric infrastructure, because the framework-like basic functionality provided by generic methods has to be individually extended for each type of the CD4A model.

To demonstrate this pattern, the example in Figure 7.2 is used. Assuming that class `C` is responsible to display different instances of class `A` depending on their run-time type. Therefore, the `display(A a)`-method is provided. When it is invoked, the call is dispatched to the `a` instance passed as an argument, i.e., `a.accept(this)`. At run-time, `a` can either be an instance of `CA` or `SA`. Hence, the `accept(C c)`-method of the corresponding class is executed. This method dispatches the call to the `C` instance and passes itself as an argument, i.e., `c.doDisplay(this)`. Depending on the run-time type of the argument, either the `doDisplay(CA o)`-method or the `doDisplay(SA o)`-method is invoked.

However, the Double Dispatching-Pattern has some restrictions, which are explained using the example in Figure 7.2. In particular, it requires that a `doDisplay()`-method for every subclass of `A` is provided in the `C` class. In particular, `C` needs to know the complete `A`-hierarchy and is has to be adapted for each `A` subclass. Otherwise, double dispatching will fail. Another pitfall is the parameter's type of the `doDisplay()`-

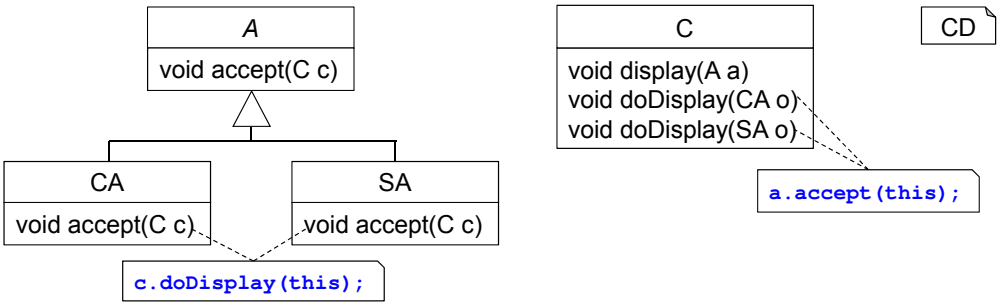


Figure 7.2: An example of the Double Dispatch-Pattern used to ensure type-safety.

methods. If the parameter type is to generic, e.g., `Object`, then double dispatching will fail, because this generic type always applies.

7.1.3 Run-time Environment and Modularity

In general, not every part of a software system needs to be generated to reduce complexity of the code generator (cf. Section 3.2). For example, source code that is independent of the input model but required by the generated source code. This non-generated part is considered as a model-independent "Platform Layer" [SVC06]. In the data-centric infrastructure, the model-independent non-generated part is statically defined and placed in a *RTE*, which is used by the generated source code. Hence, a RTE is defined as follows:

Definition 17 (Run-time Environment (RTE))**.** *A RTE is a set of predefined artifacts used by the generated source code to fulfill its requirements and at the run-time of the product.*

A RTE can be seen as an external library required by the generated source code, e.g. Java library. It is created by a generator developer (cf. Section 3.1.3) during code generator development and deployed with the generated application. However, it is crucial that a RTE does not depend on the generated classes, because this will result in non-compilable RTEs as the dependency can only be resolved after the code generator has been executed. Hence, a RTE has to be either self-contained or may depend on other statically available source code, e.g., external Java libraries.

The design of a RTE may influence the use of the generated software system. For example, if the RTE is designed such that the generated software system is only functional as a whole, modularity - "the property of a system that has been decomposed into a set of cohesive and loosely coupled modules" [BME+07] - of the generate source code is hampered. Here, we consider each module as a set of artifacts (from the RTE and the generated source code) to realize a particular functionality that can mostly be used standalone. To support MDD of data-centric applications (cf. Section 3.3), a data-centric

infrastructure and RTE have to enable standalone use of certain predefined parts. For instance, the application layer representing the data structure can be used without the presentation layer representing a framework-like use of a data-centric application.

7.2 Mapping CD4A Models to an Application Layer

The application layer represents the data structure, which is mapped from a CD4A model described in Chapter 5. Additionally, to respect the general considerations and architectural design drivers (cf. Section 7.1), the application layer is extended with consistent object instantiation and efficient management of the instantiated objects.

In the remainder of this section, the systematic mapping of these application layer extensions from CD4A models are presented.

7.2.1 Object Instantiation and Manipulation

The CD4A-to-Java mapping is adapted to a package visible constructor for each Java class representing a CD4A class to ensure that no object can be instantiated using `new`-statements. Otherwise, such object instantiation would conflict with handwritten customizations concerns (cf. Chapter 6). In more detail, whenever an object is created using the `new`-statement, handcoded extensions can only be realized by adapting each instantiating statement in the generated source code.

A common approach to handle object instantiation is the *Factory*-Pattern [GHJV95], which has been proposed to handle object instantiation (cf. [HBR00, MSHL06, Rum12]). It, however, may lead to the telescoping anti-pattern [NS16] and does not handle updates of already created objects. Hence, in this thesis, object instantiation is handled by the Builder-Pattern [GHJV95]. Such builders exist for each concrete and abstract CD4A class and provide mutators for each attribute and association as well as a `build()`- and an `isValid()`-method to create instances and validate if the instance to be created does not violate data consistency. In addition, builders are also used to modify existing instances. Hence, the builder provides an `over()`-method that allows to create a builder for a particular existing object. Each change of the object that does not violate data consistency is automatically persisted on `build()`-method invocation.

A builder for abstract classes is primarily intended for handwritten extensions, e.g., to instantiate subclasses not present in the CD4A model. Hence, by default it throws an exception whenever the `build()`-method is called. Moreover, for CD4A interfaces and enumerations no builders are needed, because no instance can be created.

In addition, whenever a CD4A class has a superclass, the builder respects this hierarchy by reusing existing functionality of its superclass builder, i.e., extending the builder of the superclass.

This systematic mapping of the Builder-Pattern is shown by example in Figure 7.3. For the attribute `String name` (l.2 at the top), the mutator `setName(String name)`

exists. Likewise, the mutator `setAccount(Map<Long, Account> o)` for the association in l.5 (at the top) is generated. In addition, an `addAccount(Account o)`-method is generated for this association to add individual links. Note that the key can be omitted in the method signature, because a name qualifier is used. The `init()`- and `get()`-method realize the Override-Static-Pattern (cf. Section 6.1.2). Furthermore, the `isValid()`-method is protected, because it is only used internally and should not be accessible when using the builder to avoid data inconsistencies. The additional `build()`-method instantiates an object, i.e., the `ConsultantImpl` instance in Figure 7.3. However, before an object is instantiated, the `isValid()`-method is called to ensure that the object itself and objects affected by the instantiation will be consistent. Otherwise, the `build()`-method throws a `DataStructureViolationException` is thrown.

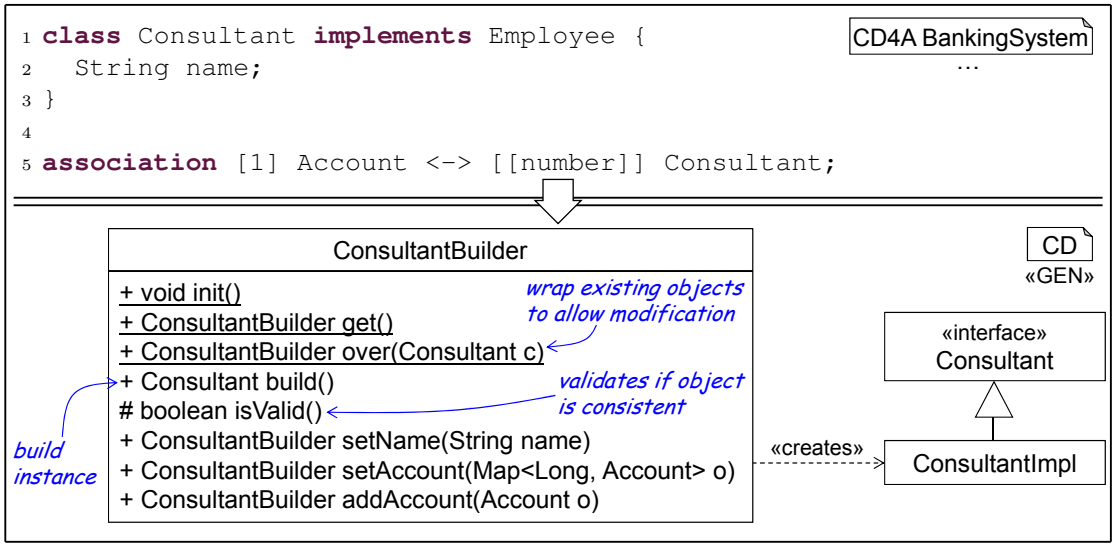


Figure 7.3: An example of mapping a CD4A class (at the top) to a Builder-Pattern implementation (at the bottom).

The technical realization of the `build()`-method is shown in Listing 7.1. Before an object is created, the method checks if data consistency is ensured (l.2). The implementation of the `isValid()`-method is explained in the remainder of this section. If data consistency is ensured, the method checks whether the builder is used to change an already existing object or to create a new object (l.3). Therefore, each builder contains a variable to associate an already existing object, i.e., the `object` variable of type `Optional`. If the an already instantiated object is associated, the builder updates all attributes (l.4). Alternatively, if the builder is used to create a new instance it uses the protected constructor of the class (l.6) and sets all attributes in the constructor. Afterwards, all associations are updated (ll.9-10) and the instance of the object is returned (l.11). However, if data structure is violated an exception is thrown (l.13).

```java
public Consultant build() {
  if (isValid()) {
    if (object.isPresent()) {
      object.get().setName(this.name);
    } else {
      Consultant newObj = new ConsultantImpl(name);
      object = Optional.ofNullable(newObj);
    }
    account.keySet().stream().forEach(x ->
       object.get().addAccount(account.get(x)));
    return object.get();
  }
  throw new DataConsistencyViolationException("0xD7260: ... ");
}
```

Java «GEN» ...

Listing 7.1: Implementation of the `build()`-method for Figure 7.3.

To demonstrate how data consistency is ensured by the builder, the implementation of the `isValid()`-method for the example in Figure 7.3 is shown in Listing 7.2. The method checks if all attributes are set (l.2). Furthermore, because in this CD4A model, the qualified association has cardinality `[1]`, the method checks that for all existing keys an association link exists (ll.6-9). If none of the checks evaluated to false, the method returns true (l.10).

```java
public boolean isValid() {
  if (this.name == null || this.name.isEmpty()) {
    return false;
  }

  for (Long key : this.account.keySet()) {
    Account col = this.account.get(key);
    if (col == null) return false;
  }
  return true;
}
```

Java «GEN» ...

Listing 7.2: Implementation of the `isValid()`-method for Figure 7.3.

Figure 7.3 also shows that a builder realizes the Override-Static-Pattern to create instances of the generated builder. To explain the use of the generated builder to create objects, we consider in Listing 7.3. In this example, an instance of the builder is retrieved by calling the `get()`-method (l.1). Afterwards, the attribute is set in l.2. By calling the `build()`-method in l.3, a `ConsultantImpl` instance is created.

```
1 Consultant c = ConsultantBuilder.get()                    Java
2                 .setName("Alexander")
3                 .build();
```

Listing 7.3: Example of using the generated builder in Figure 7.3 to create `ConsultantImpl` objects.

7.2.2 Data Structure Management

To manage the instantiated objects and encapsulate synchronization concerns when storing objects using a persistence infrastructure (described in more detail in Section 7.5), additional management facilities for each CD4A class and interface are provided as already proposed [BHKN96] (cf. *PR-3*). Each management facility is systematically mapped form a CD4A class and interface. However, no management facility for CD4A enumerations exists, because they cannot be instantiated. Moreover, management facilities for CD4A interfaces delegate the request to the implementing classes.

Such management facilities -subsequently called *managers*- allow to iterate, update, search for, and remove objects of the managed data structure. A builder for a particular type of object (cf. Section 7.2.1) uses the manager's functionality, when creating or updating objects. Hence, to ensure that only builders are allowed to add and update objects, this functionality of the manager is package visible.

The manager supports two operation modes (cf. *PR-5*). One offline mode that is primarily intended for demonstration and testing purposes without a persistence infrastructure, as explained subsequently. The other mode realizes management functionality by using a persistence infrastructure to store objects, as further explained in Section 7.5. Both modes are required to support MDP and MDD of data-centric applications.

Figure 7.4 demonstrates the systematic mapping of a manager. It shows a mapping for the CD4A class `Share` (at the top) to the `ShareManager` (at the bottom), which is the technical realization of the manager. It is initialized using the `init(Storage<Share> s)`-method to set a concrete implementation of the `Storage` interface, which realizes the actual functionality of the manager for one of the two operation modes above. In this example, the `ShareStorageMock` realizes non-persistent offline mode, whereas the `SharePersistenceStorage` realizes the online mode. Note that the `Storage` interface does not need to be generated but is located in the RTE, because it is not model-specific. All other methods provided by the `ShareManager` allow to remove an instance (`remove()`-method); iterate over all instances (`iterate()`-method); retrieve the amount of stored instances (`size()`-method); remove all instances (`clear()`-method); and search for instances with a particular value (`search(String s)`-method). As shown by example of the `remove()`-method, each method checks if one mode is selected and invokes the corresponding method of the concrete implementation for the chosen mode.

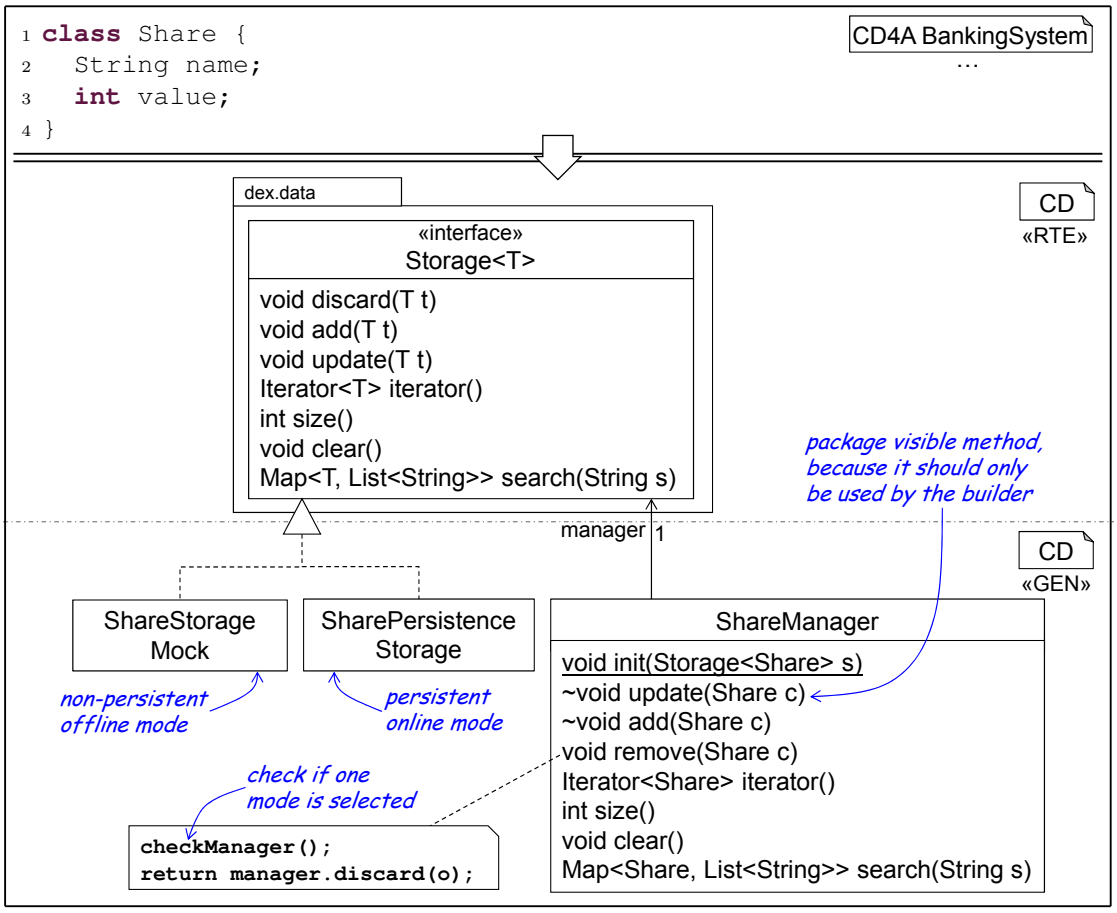


Figure 7.4: Example of mapping a concrete management facility (at the bottom) for a CD4A class (at the top).

Technically, the implementation of the offline mode (e.g., `ShareStorageMock` in Figure 7.4) uses a hashmap to manage all instantiated objects. The online mode (e.g., `SharePersistenceStorage`) uses the WebService provided by the persistence infrastructure, as explained in more detail in Section 7.4.

Mapping of CD4A Hierarchies

Whenever CD4A classes or interfaces extend other CD4A classes or interfaces, the managers regard this hierarchy to reuse existing implementation. In particular, additional dispatching interfaces to realize the Double Dispatching-Pattern (cf. Section 7.1.2) are mapped for each superclass or super interface. Each dispatching interface contains for each class in the hierarchy the manager's `add()`, `discard()`, `update()`, and

`clear()`-methods with a `do`-prefix and the subclass' type as a parameter. Moreover, for all other methods, the implementation is adapted to regard all subclasses. Note that this interface is necessary to ensure support for online and offline mode, because the implementation of the `Storage` interface changes depending on the mode.

The example in Figure 7.5 shows how hierarchies in the offline mode are handled for the CD4A class `Account` (at the top). Note that for presentational reasons the online mode has been neglected, because it is realized similarly. For the CD4A class `Account`, which has two subclasses (`SavingsAccount` and `CheckingAccount`), the additional dispatching interface `AccountStorageDispatcher` is provided. It contains a `doAdd()`, `doDiscard()`, `doUpdate()`, and `doClear()`-method for each particular subclass. In this example, only the `doDelete()`-methods are shown for presentational reasons.

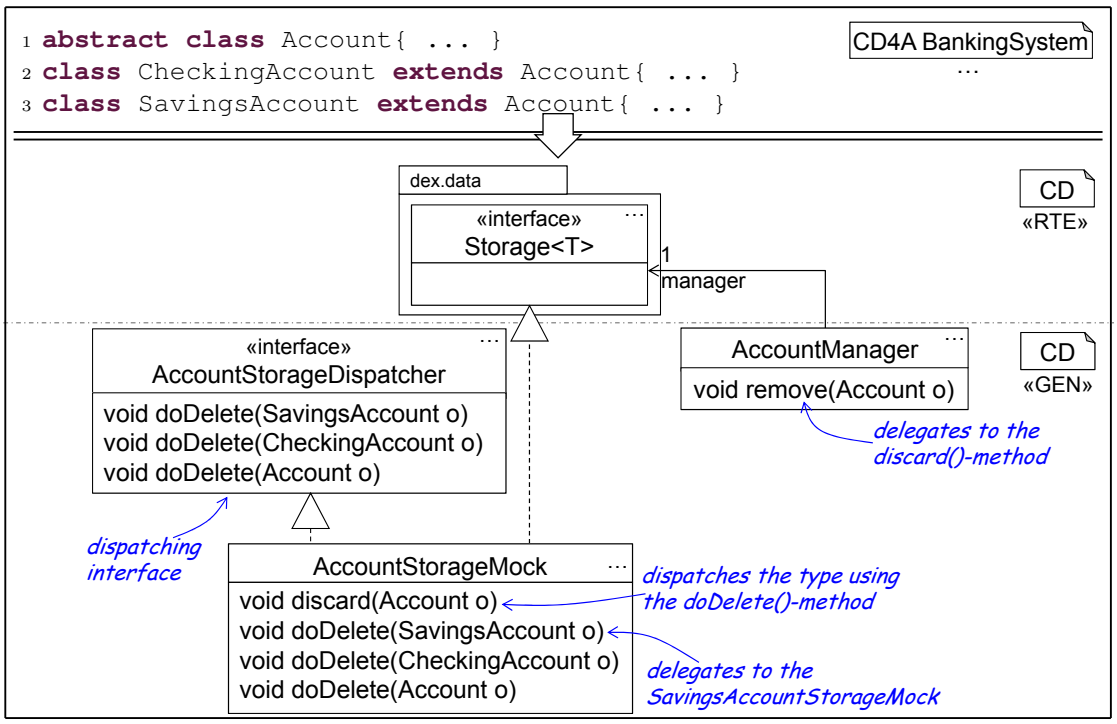


Figure 7.5: Hierarchies of CD4A classes and interfaces are respected by an additional dispatching interface that realizes the Double Dispatch-Pattern.

Whenever the `remove()`-method of the `AccountManager` class is called, it is delegated the call to the concrete implementation of one of the two modes. In this case, we assume it is the offline mode. Hence, the `discard()`-method of the `AccountStorageMock` is called. This method uses double dispatching to call the appropriate `doDelete()`-method, which delegates the call to the `remove()`-method of the manager responsible for the type of object. For example, if an instance of `CheckingAccount` is to be

deleted, the `remove()`-method of the `CheckingAccountManager` is called. Note that a `doDelete(Account a)` is added, which by default throws a `NotImplemented-Exception`. This ensures that manually-written subclasses of the `Account` class have to be handled by the developer as well.

Deletion of Instances from the Data Structure

Besides manipulating objects, object deletion may violate data consistency. For example, if the deleted object is linked to another object, the cardinality is `[1]` or `[1..*]`, and the object is the last in the list. Another example is the deletion of linked objects in unidirectional associations, where the target is unaware of the link. Although the latter case can be handled by making the association bidirectional (cf. [BFL13]), it contradicts to the semantics of unidirectional associations (cf. Section 5.2.6).

Hence, object deletion is handled by the data-centric infrastructure rather than by an object itself. An object can be deleted if

- it is not the target of a mandatory association, which by removing the association link would violate data consistency,
- it is associated by an optional attribute of some other CD4A class, or
- it is the left-hand side of a composition

In all other cases, deletion is not possible. If these preconditions are satisfied, the instance is deleted as follows:

1. Remove every association link, where the object to be deleted is the target.
2. Remove the object from every optional attribute.

Note that for CD4A composition, only the left-hand side of a composition (the whole) can be removed. If it is deleted, then the right-hand side (the parts) is deleted as well. Technically, this can be ensured via Java weak references (cf. [AHMM07]), which, however, requires to mark deleted objects as deleted while links still exist (cf. [Ges08]). In this thesis, the manager of the composition's part-of type forbids deletion and only the manager of the whole's type can delete the parts.

7.3 Mapping CD4A Models to a Presentation Layer

The presentation layer of the data-centric application provides a GUI to manage the instantiated objects (cf. *PR-2*). Mappings of UML diagrams to GUIs have already been proposed, e.g., [MSHL06, GGLVG08, LSHA08, HMZ11, Lan16], and serve as a foundation for the mapping developed in this thesis.

The main model-independent parts of the GUI are shown in Figure 7.6. Note that this screenshot shows additional model-dependent elements, which are added to the model-dependent part. The *Main Window* is the enclosing element of the GUI. It contains a *Command Area*, which provides the (S)CRUD operations, a *Tree Area*, which provides an overview of all CD4A interfaces and classes, and a *List and Edit Area*, which hosts ListViews and EditViews (cf. Section 7.1).

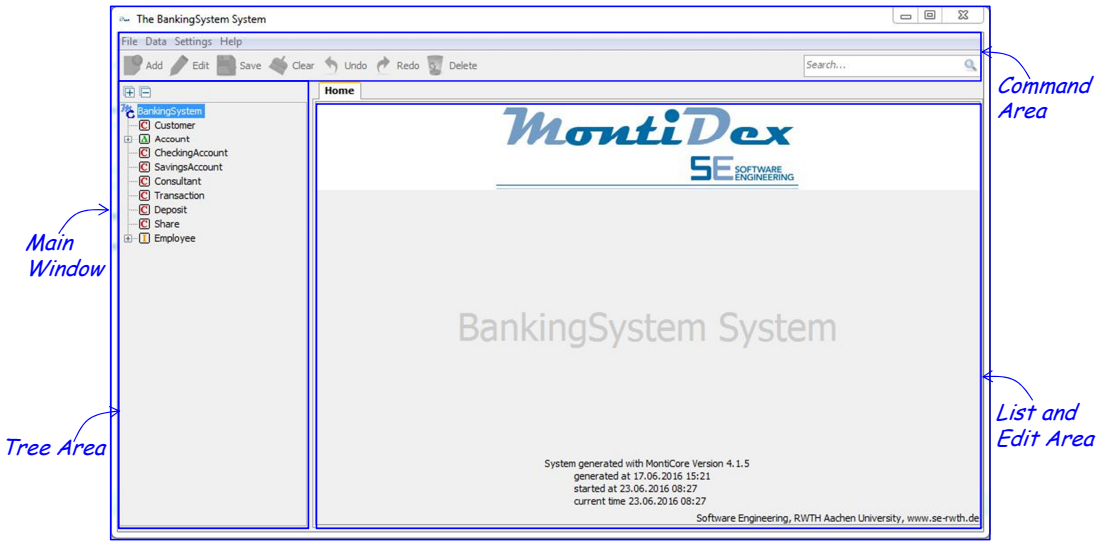


Figure 7.6: The model-independent part of the MontiDEx product GUI.

Subsequently, the systematic mapping developed to map CD4A models to model-dependent GUI elements, which extend the model-independent parts, is presented.

7.3.1 Mapping Model Definition, Interfaces, Classes, and Enumerations

Each model definition is used to structure the generated source code and ensure referential integrity, as proposed in Section 5.2.1. In addition, the model's name is used as the application's name, as shown in Figure 7.6.

A CD4A interface and class is mapped to an item in the Tree Area allowing direct access to a ListView of instances of the selected CD4A interface or class, as shown on the bottom left-hand-side of Figure 7.7. It also shows how the CD4A class `Customer` is mapped to a ListView, which displays all instances of the CD4A class and its subclasses with the attributes' values. This also holds true for CD4A interfaces. For CD4A enumerations, no list and edit capabilities are provided because they only group values (cf. Section 4.1).

EditViews allow to create new instances or edit existing ones by providing edit capabilities for each non-derived attribute and non-derived association. Depending on the

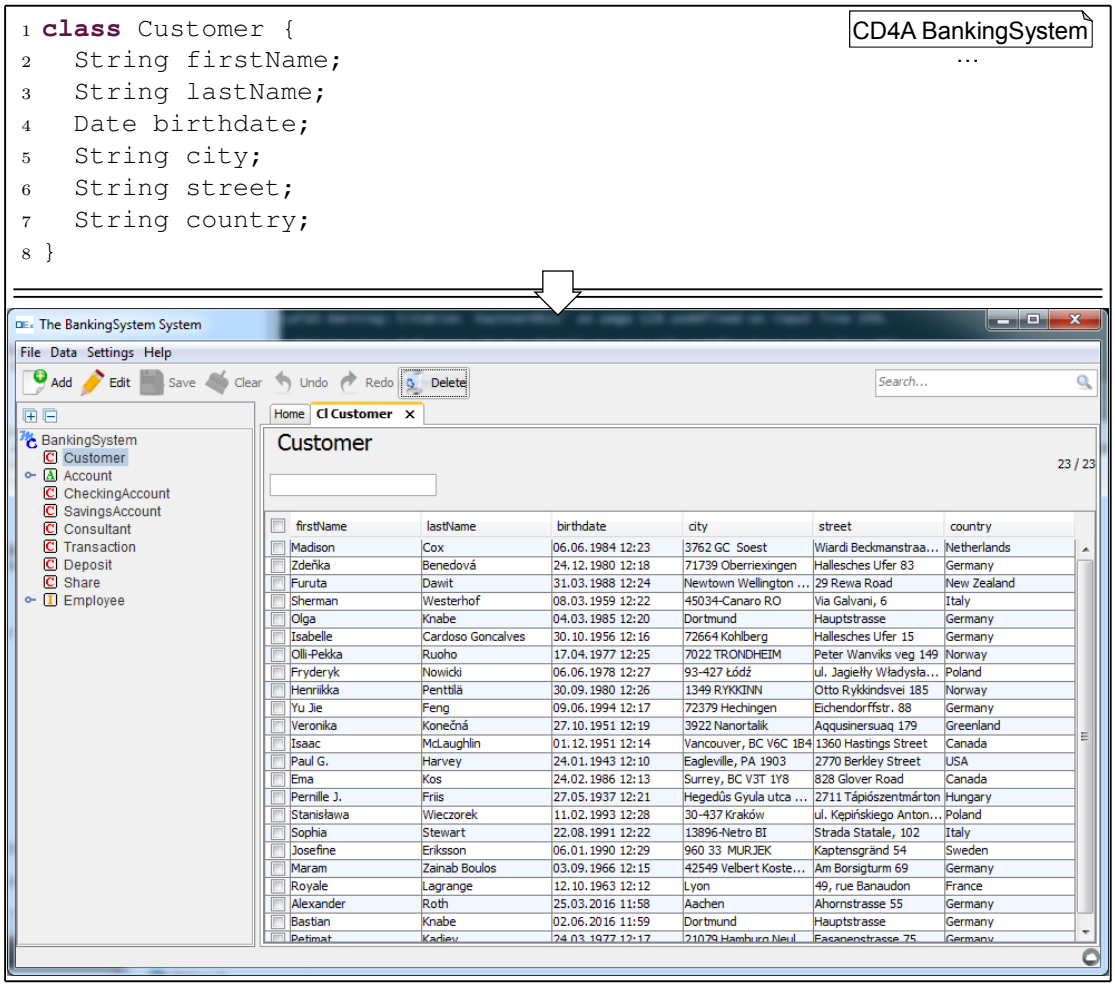


Figure 7.7: CD4A interfaces and classes are mapped to views showing their instances (right-hand-side), which can be accessed via a tree view (left-hand-side).

attribute's or the association's type, the GUI representation varies and may provide input validation, i.e., only numerical values are used for attributes with a numerical type, and the value is not empty. An overview of the GUI elements for attributes are summarized in Table 7.1 on page 124. For presentational reasons the additional label to denote the attribute is omitted and all GUI elements refer to Java Swing GUI elements.

Mapping CD4A Hierarchies to EditViews

The EditView also changes if the corresponding CD4A class has a superclass or implements an interface. Such hierarchies are represented by displaying all GUI elements of

the parents, because whenever an object in the hierarchy is edited, the parent's attributes have to be editable as well to allow instantiation of consistent objects.

In addition, the hierarchy is also represented in the Tree Area by showing the parent and child relation.

For example, Figure 7.8 presents the mapping of the CD4A model at the top to the GUI at the bottom. As depicted in the bottom-left-hand side, the hierarchy shows that the CD4A class `A` has two subclasses, namely `B` and `C`. The hierarchy of `C` is illustrated by the stack of editing elements for the `A`, `B`, and `C` in the EditView (at the bottom).

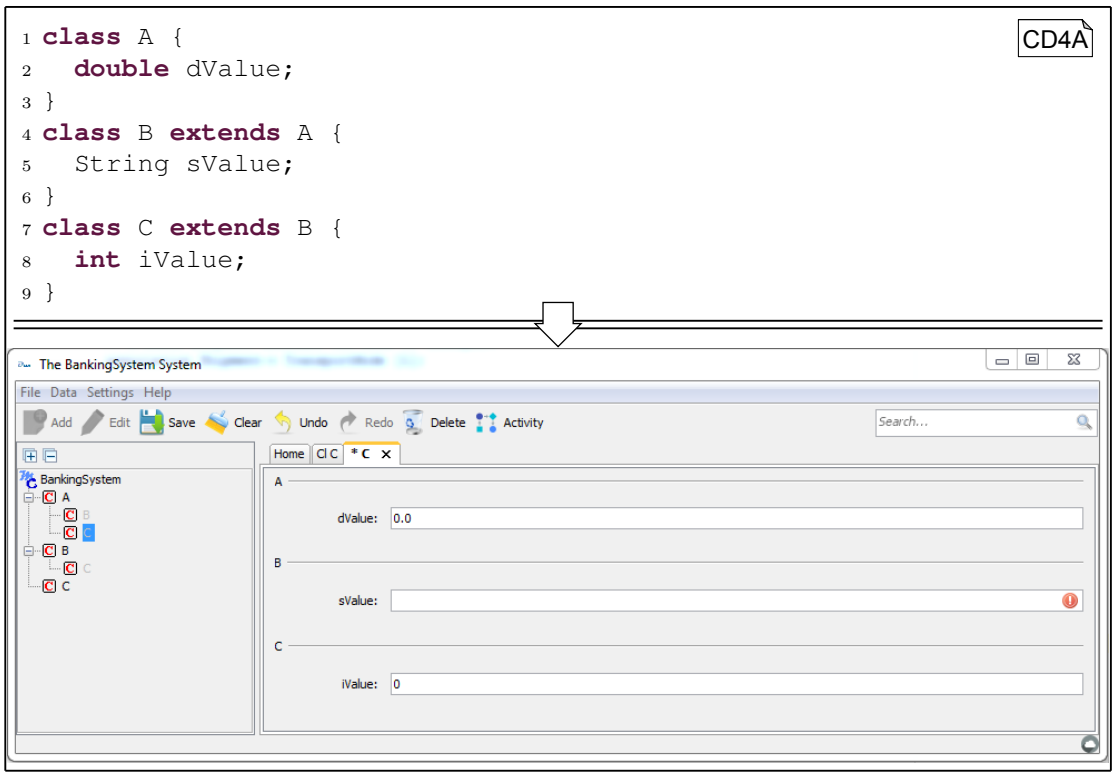


Figure 7.8: Hierarchies in the CD4A model are represented in the Tree Area (left-hand side) and in the EditView (right-hand side).

Mapping Associations to EditViews

Similar to attributes, associations are mapped to GUI elements in the EditView. In particular, each association is mapped to a list representation, which contains a set - if the association is ordered a list - of referenced instances. If the association is qualified, the list representation is split into a list for keys and a list for the key's values.

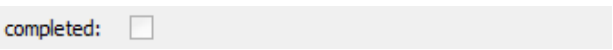
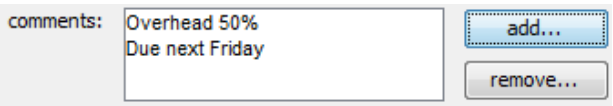
CD4A Attribute Type	GUI Element
• CD4A interface or class as a type (e.g., `B b;`)	Non-editable text field to select existing instances. 
• Primitive or wrapper type (e.g., `String name;`)	Editable text field. 
• Enumeration type (e.g., `CEnum day;`)	A ComboBox storing all values of the enumeration. 
• External data type (e.g., `Date date;`)	By default a non-editable text field, because it has to be implemented by hand. However, for the `Date` type a Date-Picker is predefined. 
• `Optional` type (e.g., `Optional<Date> date`)	A CheckBox and a representation for the generic parameter type. The CheckBox enables and disables edit functionality. 
• Derived attribute (e.g., `/boolean completed`)	The graphical representation of the attribute's type is used but is non-editable. 
• Collection type (e.g., `List<String> comments`)	A list to represent all added values and capabilities to add and remove values. 

Table 7.1: Overview of the mapping of attributes to GUI elements.

For each association, representation it is allowed to add, show, or delete an association link. If an association is derived, only the show action is allowed. Moreover, because a list representation allows to add multiple elements, the cardinalities `[1]` and `[0..1]` are supported by restricting the set of elements to be added. In case an association is mandatory and not yet set, the GUI element is colored in red to provide user feedback. In addition, derived associations are highlighted by a blue color.

Another example, shown in Figure 7.9, illustrates the mapping for associations. The association in l.4 (at the top) is mapped to a split view, because it is a qualified association, which requires a key and a value. Moreover, the association in l.5 (at the top) is mapped to the GUI element with the blue-colored list to denote a derived association. Note that the edit buttons are disabled. Finally, the association in l.6 (at the top) is mapped to the red-colored association to illustrate that it is a mandatory association and a value is not set.

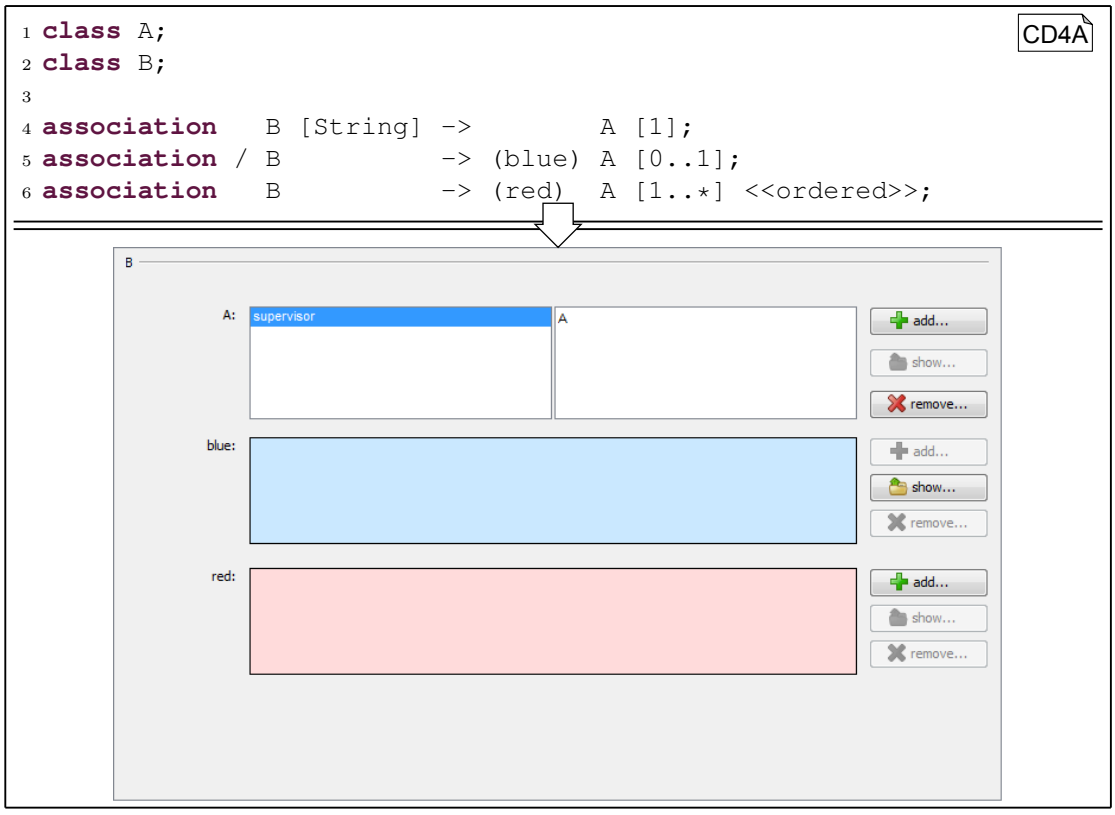


Figure 7.9: Associations are represented as lists of elements, where blue-colored denote derived associations and red-colored denote mandatory associations.

The chosen coloring denotes input validation for associations only in terms of existing values, i.e., an association link has to be assigned, because it is mandatory with respect to the CD4A model. However, by setting an association link, data consistency is not ensured, because the GUI only represents the values but does not check if, e.g., association links can be set (cf. *PR-2-3*). This is done by the corresponding builder, when attempting to save the changes.

7.3.2 Technical Realization of GUI Architecture

The proposed infrastructure is based on common design principles and patterns for implementing GUIs [Kar08] and separated into a RTE (cf. Section 7.1.3) and into a model-specific part (*GEN*), which is generated from CD4A models. The «RTE» and «GEN»-abbreviation are used in the remainder of this section to separate parts of the data-centric infrastructure.

To manage the main application and creating particular ListViews and EditViews for each CD4A class and interface, the Application Controller-Pattern [Fow03a] is used. It is realized by the `AbstractController`, which is part of the RTE, as shown in Figure 7.10. Furthermore, the `MainWindowPresenter` and `MainWindowView` realize the *MVP*-Pattern [Kar08] for the Main Window and are located in the RTE as well, because it does not need to be generated (cf. Section 7.3). Likewise, the `MainWindowContent`, `MainWindowMenuBar`, and `MainWindowToolbar` realize the view of the Main Window, the Menu Bar, and the Toolbar.

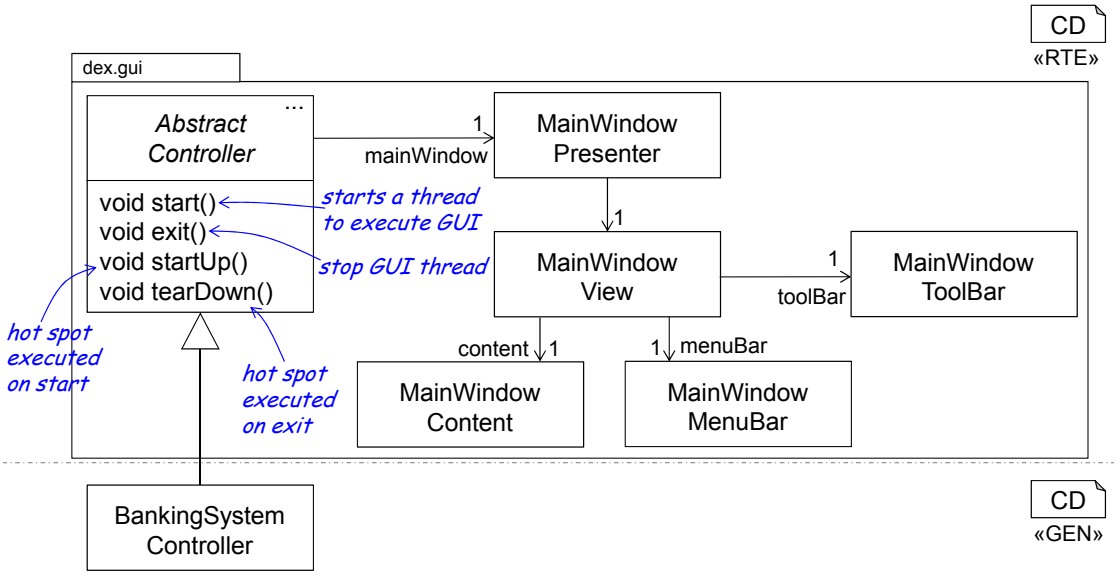


Figure 7.10: Overview of the main architecture of the MontiDEx product GUI.

For each CD4A model, a concrete subclass, e.g., `BankingSystemController` in Figure 7.10, of the `AbstractContoller` class is required. It specifies for each CD4A class and interface the ListViews and EditViews that have to be instantiated at run-time. At run-time the `AbstractController` uses double dispatching to open the object type's ListView or EditView. The main methods of the `AbstractController` are:

- **`start()`:** Launch a thread to execute the GUI.
- **`exit()`:** Immediately terminate the GUI. If the online mode is used, the currently logged in user is logged out and the connection is terminated.
- **`startUp()`:** At start up, this method is called. By default it is empty and represents a hot sport for handcoded extensions (cf. Section 6.3).
- **`tearDown`:** When the `exit()`-method is called, this hot spot is called to allow handcoded extensions at shut up. By default this method is empty.

Technical Realization of EditView

Each ListView and EditView is realized using the MVP-Pattern, which is CD4A model-specific. In contrast to ListViews, EditViews comprise multiple MVP-Patterns to support CD4A inheritance as described in Section 7.3.1. Therefore, each EditView is a composition of multiple views.

To demonstrate the mapping of EditViews from CD4A models, the example in Figure 7.11 is used. It shows the mapping of the CD4A class `CA`. The `CAEditPanelView`, which represents the view, is composed of the `CAPanelComponent View` and the `APanelComponentView`, because the `A` class is the superclass of the `CA` class (at the top of Figure 7.11). The abstract superclasses located in the RTE provide template methods and hook methods realizing the static functionality (cf. Chapter 6). Note that the technical realization of the ListView is only one MVP-Pattern per CD4A type (it is omitted here for presentational reasons) to represent the list overview, where hierarchy concerns are handled within the presenter.

The `AbstractEditPanelPresenter` offers the following methods:

- **`doAttachListeners()`:** This method is by default empty and serves as a hot spot to attach listeners to different GUI elements.
- **`execute(Command c)`:** Changes to the `AbstractEditPanelModel` are done via the model-specific commands, as explained in Section 7.3.3. The `execute( Command c)`-method executes commands of a concrete generated presenter. This execution uses the thread management facility explained in Section 7.3.4.

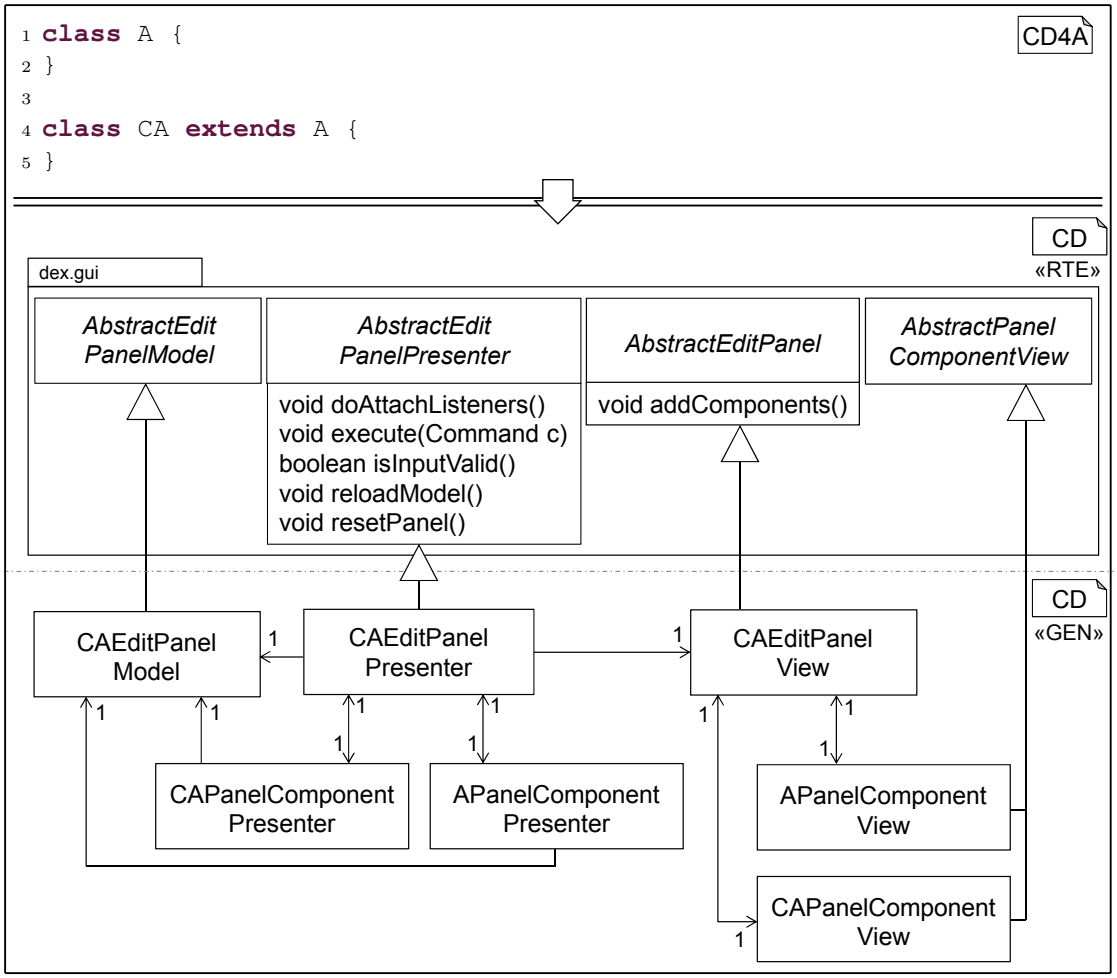


Figure 7.11: The technical realization of the EditView for the CD4A class CA.

- **isInputValid():** Each presenter is responsible for validating the GUI, i.e., checking if values for attributes and mandatory association links are assigned. This method is implemented by a generated CD4A-specific subclass. It returns true if all inputs are valid. Otherwise, it returns false and highlights the GUI elements that are invalid.
- **reloadModel():** This method updates the view by removing all displayed values and reload them from the model.
- **resetPanel():** This method clears all values currently shown in the view.

Each view for an EditView implements the `AbstractEditPanel`, which is responsible to display GUI components. It provides the `addComponents()`-method, which is implemented by the generated subclass using the mapping in Table 7.1.

7.3.3 Manipulating Objects via Model-Specific Commands

User interactions are detected by the EditView's presenter, which executes a model-specific command for each user action to change the values stored in the model. Such model-specific commands offer undo and redo functionality (cf. *PR-2-1*) and are realized by the *Command*-Pattern [GHJV95]. Each command encapsulates modifications to a particular type of objects of the data structure and is CD4A-model-specific. Hence, such CD4A-model-specific commands are systematically mapped from a CD4A model as subsequently explained.

For each attribute and association of a CD4A class, the command's name consists of an `Update`-prefix followed by the name of the CD4A class and either the name of the attribute or the name of the association as a suffix. Note that since derived attributes cannot be changed, there are no commands for them as well as for derived associations. If a CD4A class implements a CD4A interface, commands for the implemented CD4A interface exist as well. This approach is used to ensure type-safety, because the commands manipulate different models in the MVP-Pattern. Hence, multiple commands may exist for one CD4A class. Each command is executed by the presenter in the MVP-Pattern to manipulate the model.

Depending on the type of the attribute and the associations, each command stores different but statically typed values, which include the old value and the new value. It also respects the order of ordered associations.

Figure 7.12 shows an example for the mapping of model-specific commands for the `A` CD4A class depicted at the top. For each attribute and association, a command is created, which implements the `Command` interface located in the RTE. For the attribute `String name` the command `UpdateAname` is provided. It is executed by the `AEditPanelPresenter` to change the corresponding attribute or association value of the `AEditPanelModel`. This approach has been chosen, because in a client-server architecture changes may not directly be persisted after each user action and only the updates of the object should be transmitted to the server rather than the object itself (cf. Section 7.5).

When the model has been updated, the `ABuilder` is called to ensure data consistency and update the object (cf. Section 7.2.1).

In addition, to support undo and redo functionality a `history` association links all executed commands. Links are added to this association by the `AEditPanelPresenter`, when a command is executed.

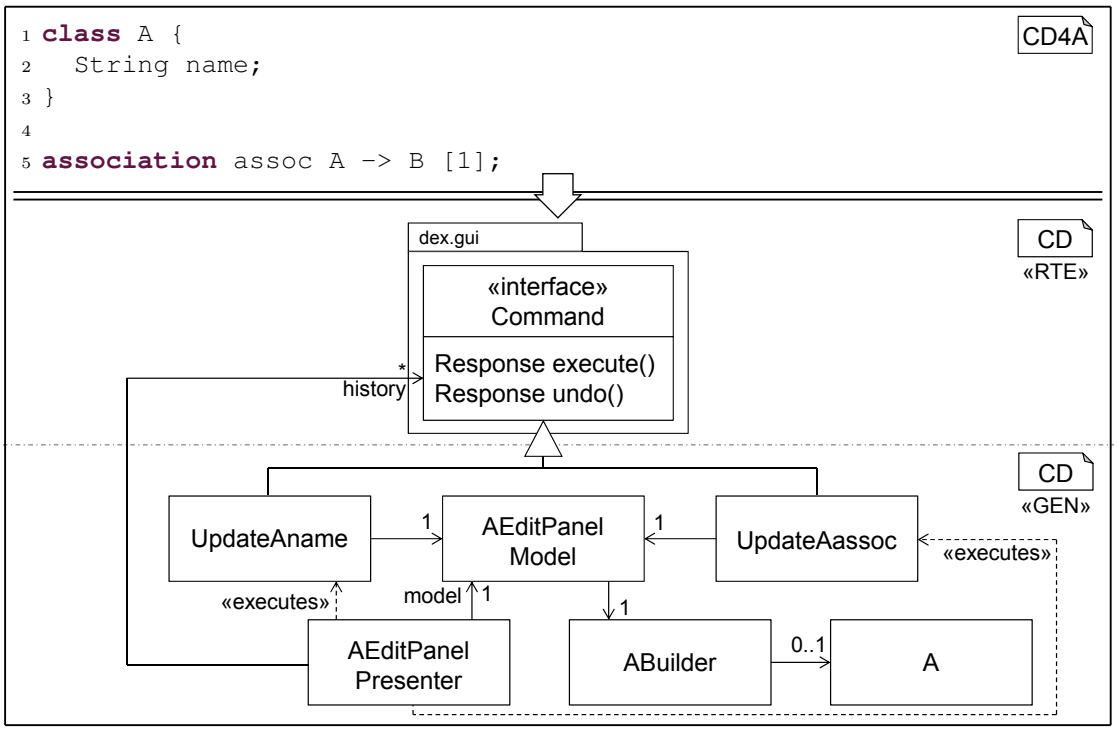


Figure 7.12: Example of mapping model-specific commands (at the bottom) for the `A` CD4A class (at the top).

Technical Realization of Commands

To demonstrate the technical realization of commands, an example of the implementation of the `UpdateAname` in Figure 7.12 is shown in Listing 7.4. The implementation is associated with a particular model (`AEditPanelModel` in l.2). For the `String name` attribute, the Java variables in ll.3-4 store the value to be set (`String newValue` in l.3) and the value that has been previously set (`String oldValue` in l.4). Moreover, the `execute()`-method sets the new value (ll.11-14) and the `undo()`-method (ll.16-18) reset the old value.

7.3.4 Managing Execution of Model-Specific Commands

Model-specific commands are executed in threads, which are managed in a centralized way by a thread management facility to ensure a responsive GUI (cf. *PR-2-2*). Such UI threads are executed in batch processing mode, i.e., for each GUI element only one thread at a time is allowed. Otherwise, updates of the same GUI element may cause run-time exceptions. Technically, this is realized as a queue of `SwingWorker`. A `SwingWorker` is a user interface thread provided by the Java Swing framework [LEW+02].

```java
public class UpdateAname implements Command {
  private AEditPanelModel o;
  private String newValue;
  private String oldValue;

  public UpdateAname(AEditPanelModel o, String v) {
    this.o = o;
    this.newValue = v;
  }

  public void execute() {
    this.oldValue = this.o.getName();
    return this.o.setName(this.newValue);
  }

  public void undo() {
    return this.o.setName(this.oldValue);
  }
}
```

Java «GEN» ...

Listing 7.4: Implementation of the `UpdateAname` command for Figure 7.12.

Each `SwingWorker` executes exactly one *Job*, which provides the implementation of the action that is executed. A Job is either generic or model-specific. Generic Jobs are provided by the RTE and only execute the `execute()` or `undo()`-method defined in the `Command` interface (cf. Section 7.3.3). Model-specific Jobs are mapped for each CD4A class only. In particular, a *Delete*-Job, which manages object deletion, and a *Load*-Job, which handles the list of currently displayed objects.

For instance, an example of the technical realization of the thread management facility is shown in Figure 7.13. The RTE consists of the `WorkerManager`, which is responsible for managing all `DexWorkers` and their execution. Each `DexWorker`, which is a `SwingWorker`, is associated with one `Job` interface, which is implemented by the two predefined Jobs `ExecuteJob` and `UndoJob`. The additional `ListPanelDeleteJob` and `LoadJob` are abstract classes implemented by the model-specific Jobs `TransactionDeleteJob` and `TransactionLoadItemJob`.

The main methods of the `WorkerManager` are:

- **`execute(DexWorker dw)`:** This method executes the `dw` Job. If a Job is currently executed, the `dw` is queued.
- **`isRunning(String s, String id)`:** To only execute Jobs of the currently active presenter, jobs are managed individually for each presenter. Hence, each Job is associated with a particular id and a presenter id. This method checks if a

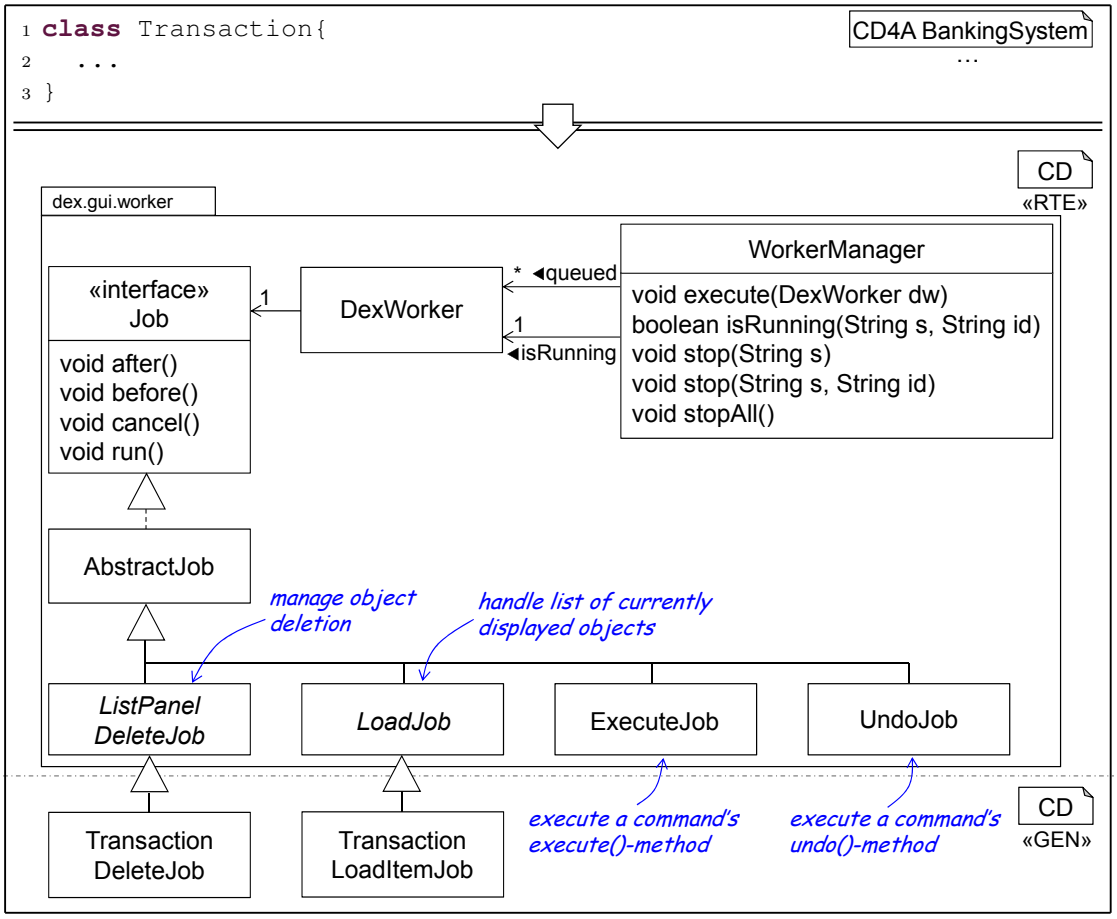


Figure 7.13: Example of mapping CD4A classes to model-specific threads.

Job with the id `id` for the presenter with the id `s` is running. If the Job with the id `id` is queued or not present, this method returns true.

- **`stop(String s)`:** This method stops all Jobs currently executed and queued for the presenter with the id `s`.
- **`stop(String s, String id)`:** Stop a particular Job with the id `id` for the presenter with the id `s`.
- **`stopAll()`:** Terminate all currently running and queued Jobs.

The `DexWorker` forwards calls to execute or stop a particular Job to the Job implementation, which is realized in the `AbstractJob` class. The main methods of the `AbstractJob` class are:

- **`after()`:** This method is executed after the Job has been finished. By default this method is empty and provided as a hot spot.
- **`before()`:** This method is executed before a Job is executed and is provided as a hot spot.
- **`cancel()`:** Cancel the currently running or queued Job.
- **`run()`:** Execute the Job. This method is called by the `DexWorker` to immediately execute the Job, because it is the first in the queue.

7.4 Generic Persistence Infrastructure

To support persistent storage for the managed data structures, a data-centric infrastructure has to provide access to a persistence infrastructure. Existing approaches for MDD of data-centric applications (cf. Section 2.4) propose a synthesis of parts of a persistence infrastructure, i.e., database schema, in addition to a data-centric infrastructure. However, such an approach hampers rapid MDP of data-centric applications, because the persistence infrastructure has to be adapted when generating a prototype. Hence, in this thesis the persistence infrastructure is generic (cf. *RE-2*).

The persistence infrastructure developed in this thesis is shown in Figure 7.14. The persistence layer of a data-centric application uses a WebService, which is provided by the persistence infrastructure, to access functionality of the generic application server. The application server realizes role-based access control and predefined business logic such as behavior when creating, updating objects, and deleting, i.e., locking all associated objects. The managed data as well as rights and roles are stored in one databases with multiple tables. Technically, a Glassfish Application Server [www16i] with a PostGres database [www16k] are used.

In the remainder of this section, the generic meta-model of the persistence infrastructure, a method to realize role-based access control, and the methods required by the data-centric infrastructure from a WebService are explained.

7.4.1 Generic CD4A Meta-Model

The server's database meta-model is shown in Figure 7.15. Its main element is an `ObjectValue`, which represents an instance of a CD4A class[1]. Each `ObjectValue` has a `type` to store the CD4A class' type. Attributes are mapped to instances of the `Attribute` class. Associations are excluded in this meta-model, because the mapping of CD4A models to concrete source code (cf. Chapter 5), maps associations to attributes.

[1] More precisely, in this context "instance of a CD4A class" means that it is an instance of the Java class mapped from a CD4A class using the mapping introduced in Chapter 5.

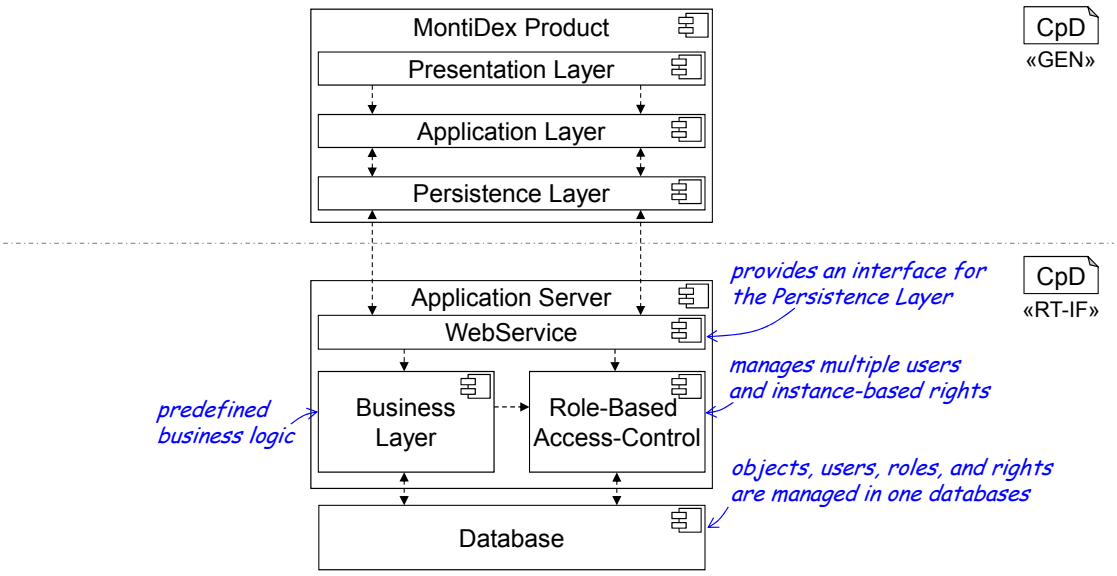


Figure 7.14: An overview of the generic persistence infrastructure.

Each `Attribute` stores the name of the attribute and associates a `Value`. To distinguish the different types of values, the meta-model provides `ReferenceValues`, which allow to reference a stored `ObjectValue` by its ID (`refId`); a `PrimitiveValue`, which stores plain text values (`value`); a `ListValue`, which represents a list of values; and a `MapValue`, which stores maps of key-value pairs. If an attribute is not a reference to another CD4A model element or a collection type (`List` or `Set`), it is mapped a `PrimitiveValue`. Furthermore, associations with cardinality `[0..1]` and `[1]` are stored as a `ReferenceValue` or as a `PrimitiveValue` depending on the referenced type. Associations with cardinality `[*]` or `[1..*]`, are stored as a `ListValue`. Finally, a qualified association is stored as a `MapValue`. An overview of this mapping is shown in Table 7.2.

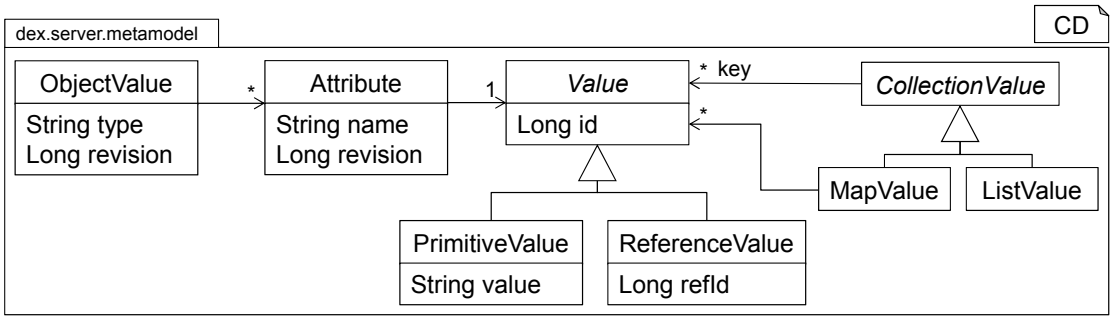


Figure 7.15: Meta-model of the generic server's database.

CD4A Model Elements	Meta-Model Element
• Attribute having a CD4A interface or class as a type (e.g., `B b;`) • An association to a CD4A interface or class and cardinality `[1]` or `[0..1]` (e.g., `A -> B [1];`)	An `Attribute` with the name of the CD4A attribute or association[2] and a `ReferenceValue` storing the ID of the associated instance.
• Attribute with an external data type (e.g., `Date s;`) • An association to an external data type with cardinality `[1]` or `[0..1]` (e.g., `A -> E [0..1];`)	An `Attribute` with the name of the CD4A attribute or association and a `PrimitiveValue`.
• Attribute with an enumeration type (e.g., `CEnum enums;`) • An association to an enumeration with cardinality `[1]` or `[0..1]` (e.g., `A -> CEnum;`)	An `Attribute` with the name of the CD4A attribute or association and a `PrimitiveValue`.
• Attribute with a `List` or `Set` (e.g., `List<String> a;`) • An association with cardinality `[*]` or `[1..*]`.	An `Attribute` with the name of the CD4A attribute or association and a `ListValue` with elements depending on the generic parameter type or the targeted association end's type.
• Qualified association (e.g., `A [String] -> B;`)	An `Attribute` with the name of the CD4A association and a `MapValue` with elements depending on the targeted association end's type and the keys depending on the qualifier type.

Table 7.2: Mapping overview of CD4A model elements to the meta-model elements.

The meta-model contains a `revision` attribute for the `ObjectValue` and the `Attribute` class. It holds the current revision of the stored instance. The revision of the `ObjectValue` is increased each time one of its attributes is changed, whereas the revision of an `Attribute` is increased if the attribute's value changes. This information is exclusively added to enable multi-user management, as described in Section 7.4.2.

7.4.2 Multi-Tenancy and Role-Base Access Control

The backbone of each InfoSys is a persistence infrastructure providing access to the stored data to different users or other systems (cf. *RE-2-1*). Its essential task is to

[2]The name of an association is understood as described in Section 4.2.4

manage different users having different roles and possibly belonging to different *tenants*, each of which is a set of users that use the same software application. In such a case, it has to be ensured that each data-centric application using the persistence infrastructure has only access to its own data for which it is allowed to define (S)CRUD operations. Hence, each data-centric application is considered as one tenant.

A variety of approaches for realizing role-based access control exist, e.g. [Ros97, TS98, LSM+98, EKBM+03, YT05, FKC07, KCW10, Gol11, JSK12, RDJK15], each of which addresses different requirements such as performance, flexibility, or scalability. In this thesis, the technical realization of role-based access control is founded on the MR-RBAC approach [TLS13, TSL15], which is primarily designed for collaborating environments. It respects *data integrity*, which ensures that each modification of the stored data is protected by access control, as well as *information confidentiality*, which ensures that each read operation is protected by access control.

An overview of the technical realization is shown in Figure 7.16. Each `Project` represents one generated data-centric application. Technically, this is achieved by a unique *identification number*, which is transmitted when the client logs in and is assigned by the generator unequally in the generation process. A `Project` manages multiple users (`User` class), each of which can have multiple roles (`Role` class) and permissions (`Permission` class) for a particular `ObjectValue`. As a result, this approach enables instance-level CRUD operations. A `RoleAssignment` is used to avoid the "role-explosion problem" [KCW10, JSK12].

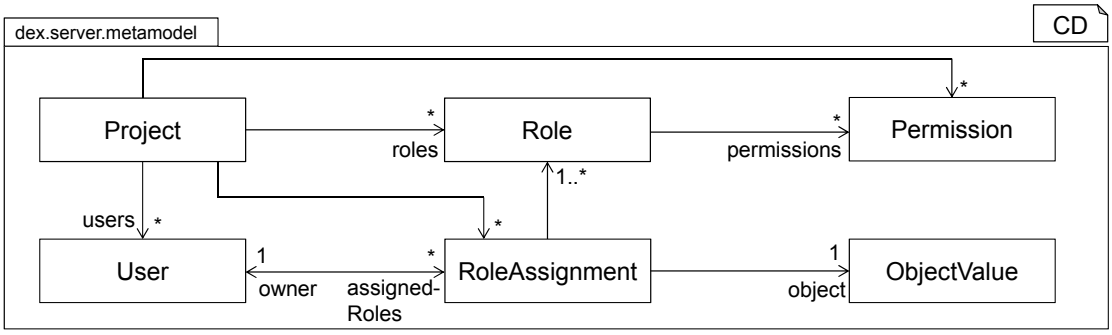


Figure 7.16: Technical realization of the adapted MT-RBAC approach for role-based access control in a multi-tenant environment.

Each permission is defined with respect to an instance with a particular type. Because the persistence infrastructure is unaware of the types, the persistence layer transmits the set of CD4A class types on the first connection. For each transmitted type, CRUD permissions are created for the default *ProjectAdmin* role, which is assigned to the first logged in user. Technically, the Apache Shiro [www16d] framework is used to realize rights and roles management It allows to define rules, which are regular expressions that

are evaluated for each request, to define permissions. For instance, for a class `A` and the user `Alex`, the expression "A:Alex:READ" defines that the user "Alex" is only allowed to read instances of type "A". Another example is the "*:Alex:*" rule, which allows all CRUD operations for the user "Alex" on every type.

Although the technical realization allows to define more detailed CRUD operations, e.g., for attributes or associations, in this thesis this kind of fine-grained role-based access control has been omitted, because it may violate data consistency. For example, if an end user is not allowed to access objects of a mandatory association, the client application cannot create a consistent object.

When supporting role-based access control, concurrent manipulation of the same object has to be addressed. In other words, two or more users should be allowed to concurrently modify the same object (cf. *RE-2-1*). A solution is to immediately transmit consistent changes to the server. Another solution is to transmit a set of changes. Whereas the primer solution minimizes the possible conflicts because only one transmitted change may conflict, the latter solution requires an approach for conflict detection within multiple changes. Hence, the primer solution has been chosen in this thesis. Technically, the approach is realized using the revision number for each `ObjectValue` of the meta-model (cf. Section 7.5). Changes are only accepted, if the object has not been changed by another user, i.e., the version number has not been incremented. Otherwise, the changes are not accepted and the end user is informed. This approach has been chosen, because it has proven to be a lightweight solution to support multiple user and distribution of data (cf. [ADH+09]).

7.4.3 Technical Realization of Accessing the WebService

To demonstrate the services provides by the persistence infrastructure, we consider the `ServerAccess` class that is provided by the RTE of the data-centric infrastructure to use the WebService. An overview of the main methods are shown in Figure 7.17. These methods can be executed by end users with the ProjectAdmin role directly from the client application.

The main methods for managing users, roles, and permissions are:

- **`login(String u, String p)`:** Log in a user with the name `u` and the password `p`. If the user does not exist or the password is wrong, false is returned.
- **`logout()`:** Log out the user currently logged in and close the connection to the persistence infrastructure.
- **`registerUser(String u, String p)`:** This method can be used to register a new user with the name `u` and the password `p`. If the user is already registered, false is returned. By default, if the user is the first registered user, he receives the

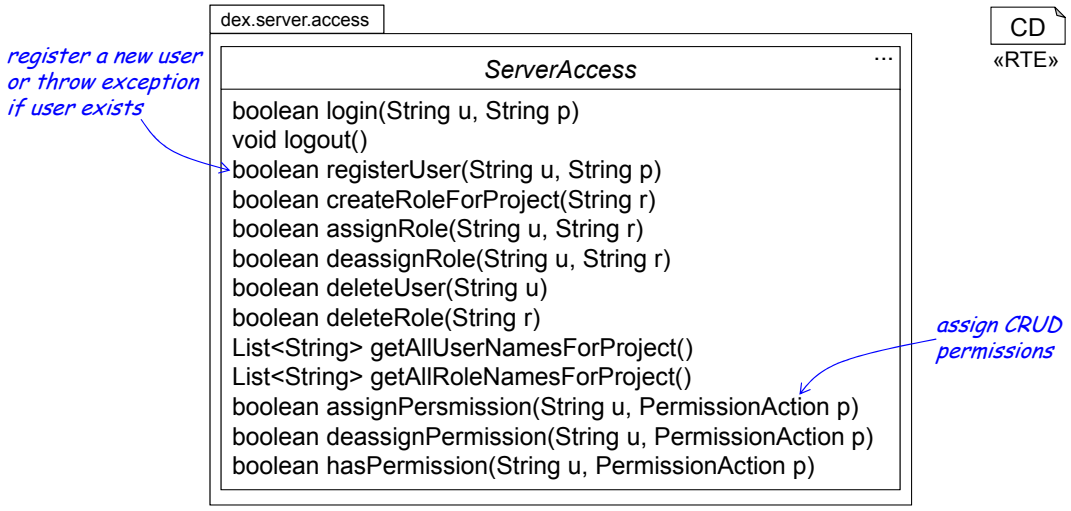


Figure 7.17: Overview of the `ServerAccess` class that handles communication to the persistence infrastructure.

ProjectAdmin role with all rights and roles assigned, as explained in more detail in Section 7.4.2.

- **`createRoleForProject(String r)`:** Create a new role with the name `r`. It this role exists, false is returned.
- **`assignRole(String u, String r)`:** Assign a role with the name `r` to a user with the name `u`. If either the role or the user does not exist, false is returned.
- **`deassignRole(String u, String r)`:** Deassign an already assigned role with the name `r` from a user with the name `u`. If the user or the role does not exist, or the user does not have the role, false is returned.
- **`deleteUser(String u)`:** Delete the user with the name `u` from the list of currently managed users. If the user does not exists, false is returned.
- **`deleteRole(String r)`:** Delete a role with the name `r` from all available roles. If the role does not exist, false is returned.
- **`getAllUserNamesForProject()`:** Return a list of names of all managed user.
- **`getAllRoleNamesForProject()`:** Return a list of all available role names.
- **`assignPermission(String u, PermissionAction p)`:** Assign the permission `p` to a user with the name `u`. The `PermissionAction` enumeration

allows to assign CRUD operations. If the permission cannot be assigned or the user does not exist, false is returned.

- **`deassignPermission(String u, PermissionAction p):`** Deassign the permission `p` from a user with the name `u`. If the permission cannot be deassigned or the user does not exist, false is returned.
- **`hasPermission(String u, PermissionAction p):`** Check if the user with the name `u` has the permission `p`. If the user does not exist or does not have the permission, false is returned.

In general, a design pattern to manage communication in a client and a persistence infrastructure is the *Response/Request*-Pattern, which allows bidirectional communication between a server and a client [HW03]. A request is sent by the client and contains a demand for modification of an object stored on the server, which processes the request and returns a response. Hence, the persistence layer converts the model-specific commands (cf. Section 7.3.3) to generic server requests. An advantage of requests is that only the changes are regarded but not the whole object and, hence, change detection is not necessary.

The methods provided by the `ServerAccess` to send CRUD requests and receive responses from the persistence infrastructure are shown in Figure 7.18.

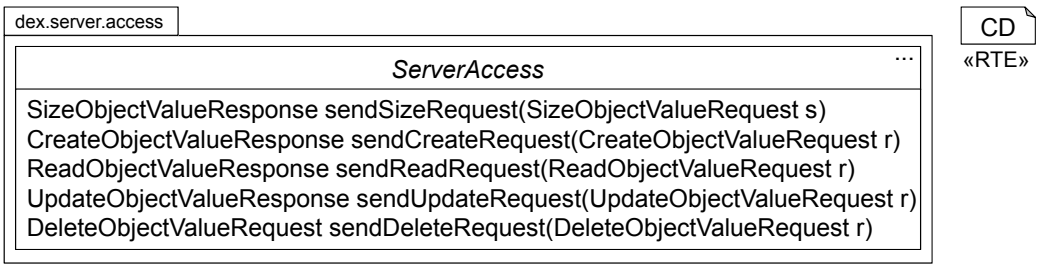


Figure 7.18: Overview of the `ServerAccess` class to send CRUD requests and receive responses from the persistence infrastructure.

All provided methods use the generic metamodel to communicate with the persistence infrastructure (cf. Section 7.4.1). They are explained in the following.

- **`sendSizeRequest(SizeObjectValueRequest s):`** Size requests realizes the search functionality and requests for a particular CD4A type the size of stored instances. The response is an integer value.
- **`sendUpdateRequest(UpdateObjectValueRequest r):`** With this method, a request to update an already stored objects is send to the persistence infrastructure. The request contains the list of attributes that have to be updated, the

objects id, and the revision number (cf. Section 7.4.1). The response reveals if the update has been successful.

- **`sendCreateRequest(CreateObjectValueRequest r)`:** This method executes a request to store a new object. It requires the object's type and a list of all attributes. The response contains the `ObjectValue` that has been stored. However, the response is empty if an error occurred.
- **`sendDeleteRequest(DeleteObjectValueRequest r)`:** A delete request for a stored instance is send to the persistence infrastructure by calling this method. It requires object to be deleted and the revision number. The response returns if deletion has been performed successfully.

7.5 Mapping CD4A Models to a Persistence Layer

The persistence layer of the data-centric infrastructure is responsible for managing the communication with the persistence infrastructure to persistently store created objects. In addition, it supports data migration tasks, because each change to a CD4A model will result in a new data-centric application, as explained in the remainder of this section.

In the remainder of this section, we explain the mapping of CD4A models to a persistence layer in the data-centric infrastructure.

7.5.1 Lazy Loading of Objects from the Persistence Infrastructure

When the data-centric application requests a stored CD4A class instance from the persistence infrastructure, the response has to be an `ObjectValue` that can be converted to a consistent CD4A class instance. Taking the understanding of data consistency as introduced in Section 5.1, this implies that all attribute values and all mandatory association links are set. However, in a worst case scenario, this may lead to a request, where the response contains all stored CD4A class instances due to transitive mandatory associations in the CD4A model.

An approach to tackle this challenge and at the same time ensure data consistency is to use a *CD4A class instance representative*, which only mimics a consistent CD4A class instance, whenever the offline mode is used for persisting objects (cf. Section 7.2.2). It is defined as follows:

Definition 18 (CD4A Class Instance Representative)**.** *A CD4A class instance representative Y of a CD4A class instance X represents X containing only all attribute values and mandatory association links that do not reference other CD4A class instances set. For all other attribute values and associations links it is allowed to reference a placeholder denoting only the existence of a CD4A class instance.*

From this understanding, it can be concluded that it is sufficient that `ObjectValue` responses only contain the necessary values such that the client can create a CD4A class instance representative. In addition, we define that for associations with cardinality `[1..*]` or `[*]` it is sufficient to contain only one placeholder to ensure consistency, because it suffices to check if an association link is contained.

Nevertheless, a CD4A class instance representative only acts as a consistent CD4A class instance, because it does not accurately substitute its representing object due to the placeholders. Hence, the persistence layer automatically resolves all placeholders whenever their value is required.

A CD4A class instance representative is systematically derived for each CD4A class only, because CD4A interfaces are not stored and CD4A enumerations do not allow for out going associations, as described in Section 4.2.4. Technically, it is realized using an adapted *Proxy*-Pattern [GHJV95], which represents an object and defines means to load required data on demand. A proxy of an object delegates a method call to the encapsulated object.

The Proxy-Pattern used in this thesis is an adapted realization to regard handcoded extensions (cf. Chapter 6), as shown in Figure 7.19. In this example, the `A` CD4A class (at the top) is mapped to the proxy class `AProxy` (at the bottom). For all attributes and associations referencing other CD4A classes, it provides accessors and mutators as described in Chapter 5, e.g., `iteratorBs()`-method. For each such method, it first checks if the placeholder is resolved (by calling the `loadIfNecessary()`-method) and afterwards delegates the call to its superclass. By calling the superclass' method, this pattern ensures that a potential handcoded extension is used. In this case, the pattern has to be adapted by the code generator to extend the handcoded extension. Besides, if an association with cardinality `[*]` or `[1..*]` is loaded, only a chunk of association links is loaded at once. In addition, the constructor of the `AProxy` requires all attributes and mandatory associations, because it has to represent a consistent object.

An essential aspect of the proposed realization is to ensure that prospective developers are unaware of a proxy's existence. This is achieved by an additional extension of the builders in the application layer (cf. Section 7.2.1), which is a package visible builder for proxies. Its only purpose is to convert server responses into either a proxy or a real instance, e.g., the `AProxyBuilder` shown in Figure 7.19 (at the bottom). Each created proxy overrides the `hashCode()`- and `equals()`-method to ensure that the encapsulated object is used when called.

7.6 Method for Consistent Data Migration

If the CD4A input model evolves and the data-centric application has to be regenerated, the tenant identification number changes and the previously persisted objects cannot be reused. This design decision has been chosen, because the consistency of the previously

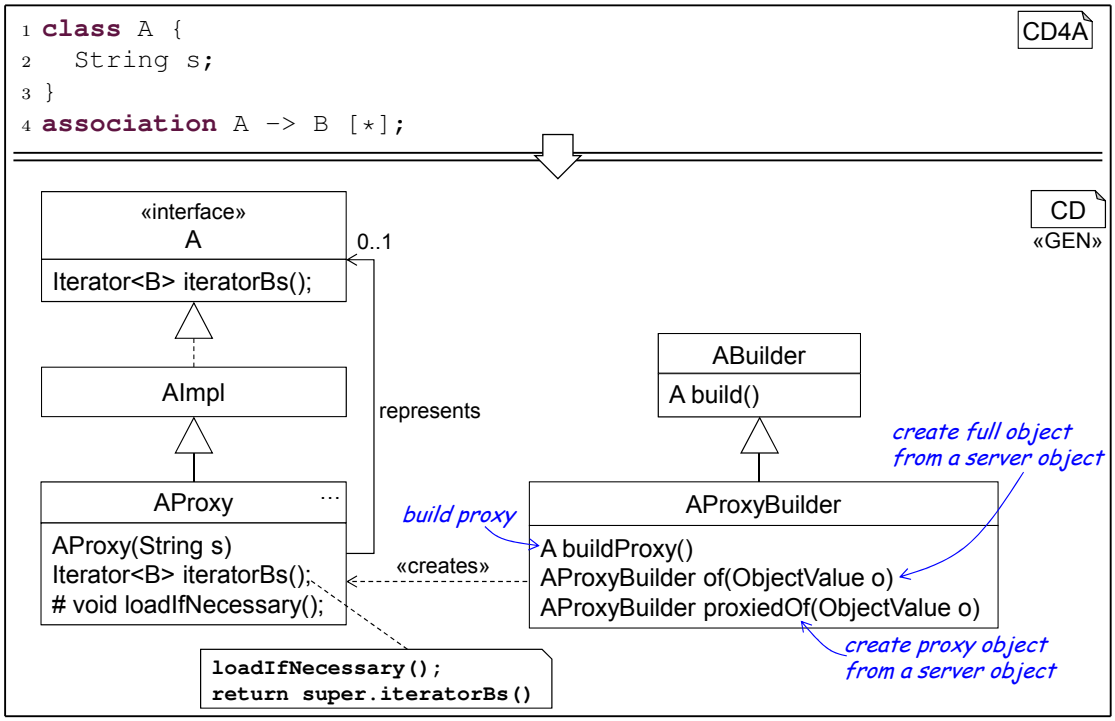


Figure 7.19: The Proxy-Pattern realization used to support handcoded extensions.

stored objects with respect to the evolved CD4A model cannot be ensured anymore. Hence, if the stored data is required, it has to be migrated to a consistent object structure of the evolved CD4A model (cf. *RE-2-3*). This process is referred to as *data migration*:

Definition 19 (Data Migration)**.** *Data migration is the process of transforming an object structure (source), which conforms to one CD4A model (source model) and is also data structure consistent, to a new object structure (target) that conforms to a different CD4A model (target model) and is also data structure consistent.*

Note that because the persistence infrastructure uses a generic meta-model, ontology evolution, e.g., [ES07], is not applicable. Moreover, data migration is only required, if the input CD4A model evolves. If the model does not evolve, regeneration to provide updates or bug fixes is always possible.

An overview of the developed data migration approach is shown in Figure 7.20. It is separated into three consecutive steps (separated by dashed lines) executed by the client. The data migration process starts by checking the rights of the user performing the migration task. Only users who are allowed to read all persisted instances of the source and are allowed to create objects of the target, are allowed to perform a data migration. If the user does not have the required privileges, the process is aborted.

Note that the user performing the migration task has to be known to the server storing the source and the server persisting the target, if different application servers are used. The first step ends by downloading the source and locally storing it as a UML/P object diagram (UML/P OD) model [Rum12, Sch12].

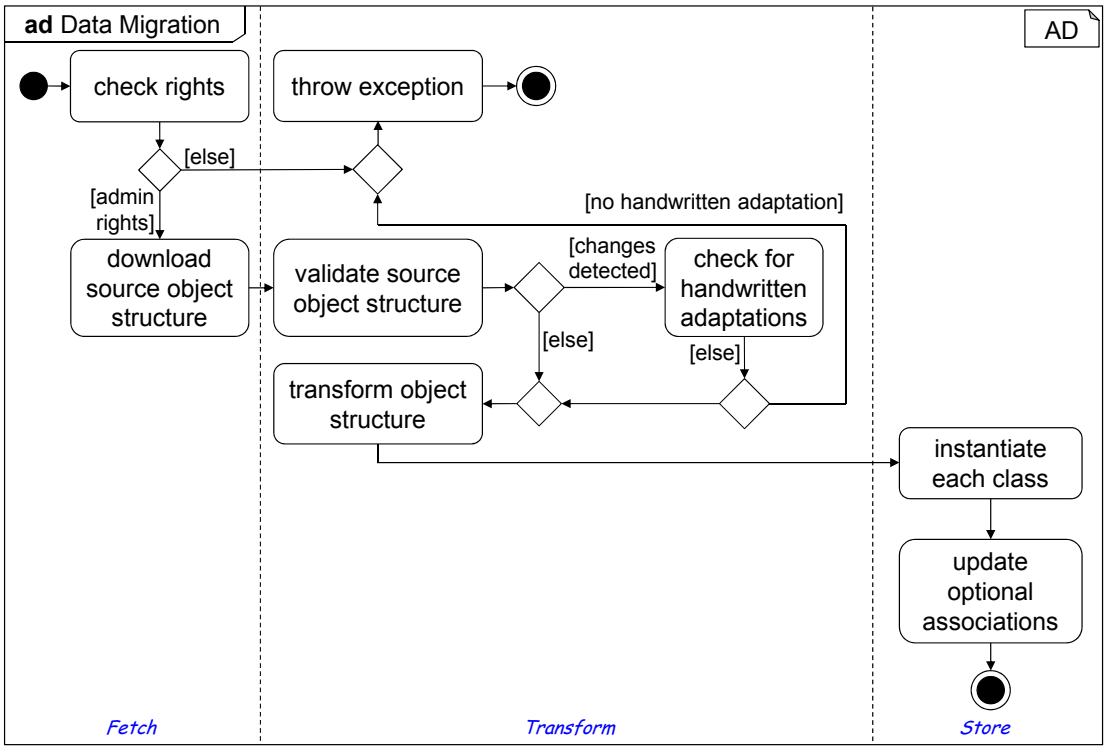


Figure 7.20: Data migration is done by downloading the all instances of the source model (Fetch), transforming it to conform to the evolved target (Transform), and storing it on the new server (Store).

In the second step, visitors traverse the stored UML/P OD to identify CD4A classes conforming to the target CD4A model, i.e., CD4A classes that did not evolve. Each CD4A class instance defined in the source but not in the target, i.e., a CD4A class instance that has been deleted, is ignored. For each evolved CD4A class instance, a manually-written adaptation of the visitor is required, which specifies how the CD4A class instance has been changed. Note that this approach is similar to specifying *Deltas* [Loo17] without employing a DSL but use Java source code instead. If no manually-written adaptation is defined for the evolved CD4A element, data migration fails. Finally, if for every evolved CD4A class instance a manually-written extension is defined, the source is transformed into the target.

In the last step, the target has to be stored on the server. Therefore, two steps are performed. First, the transformed UML/P OD is again traversed and instances of the evolved CD4A classes are created using the generated builders of the target CD4A model (cf. Section 7.2.1). However, only the mandatory associations and all attribute values are set. Second, the created objects are updated by adding the optional associations. This two-phased process is required, because bidirectional ordinary associations may introduce a cycle that makes it unfeasible to instantiate the CD4A class.

Chapter 8

Synergetic Transformation- and Template-based Code Generation

The generated data structure (cf. Chapter 5) and data-centric infrastructure (cf. Chapter 7) can be customized using the Extended Generation Gap-Pattern (cf. Chapter 6) to meet varying end user (cf. Section 3.1.3) requirements. However, the Extended Generation Gap-Pattern is not practical if the same customizations have to be applied to multiple artifacts. Moreover, it is restricted in terms of customizability, e.g., given structures in the generated Java source code are hard to be overridden. As a result, code generator reuse is hampered. To tackle these customization restrictions, the CD4A-to-Java source code mapping, which is defined by a generator developer in a code generator, has to be adaptable.

In general, extensibility and adaptability of a code generator is influenced by the approach used for code generation (cf. [ZR11b]). For example, template-based code generation can address such concerns by aspect-orientation [KR03, OH07, ZR11a, ZR11b], where adaptation is achieved by aspect weaving; or *semantically configurable code generation* [PADS12], which uses a configuration-based approach to address adaptation. Another example is aspect-orientation for transformations-based approaches [VG07] or design guidelines [HKGV10]. However, for data-centric applications, practice has shown that using one of the aforementioned code generation approaches solely has disadvantages in realization of code generator modularity (cf. [ZR11b]) and generating source code for the presentation layer and the application layer (cf. [MK09]).

Hence, an integration of (*synergetic*) transformation- and template-based code generation for object-oriented target languages has been developed in this thesis. It facilitates the following aspects:

(i) Support for flexible use of transformations and templates.

(ii) Independence of the input and target language to support reuse of transformations by using an *intermediate representation (IR)*, which is represented by a model conforming to the CD4Code ML (cf. Section 4.3).

(iii) Favor modular design of code generators (cf. *GR-1*).

(iv) Facilitate black-box integration of manually-written transformations and templates (cf. *GR-2-2*) to enable senior application developers (cf. Section 3.1.3) to adapt and extend the code generator without a detailed knowledge of the internals (cf. *GR-2*).

The goal of this chapter is to introduce the integrated transformation- and template-based code generation approach to help generator developers (cf. Section 3.1.3) in code generator design and development.

This chapter is structured as follows. First, the requirements for an integrated transformation- and template-based code generation approach that addresses code generation adaptation and extension to facilitate code generator reuse are presented in Section 8.1. Afterwards, the developed integrated code generation approach is described using a simplified SC to Java example in Section 8.2. Next, the transformation- and template-based code generation approach is extended with additional *hook points*, which are predefined spots in templates, and *template extensions* to support black-box adaptation, as described in Section 8.3. Finally, to guide generator developers in designing maintainable transformations and templates and effectively use the developed code generation approach, a method is presented in Section 8.4.

8.1 General Requirements

Transformation- and template-based code generation are general approaches that are employed in code generators to generate source code from input models (cf. Section 2.2.4). In the remainder of this section, requirements for an integrated solution of both approaches are listed. They are derived from the targeted usage scenario described in Chapter 3.

GR-1-1 Code generation modularity: Modularity of code generation is concerned with providing means to realize extensible and loosely coupled modules of the code generator [ZR11b, RR15, JHH16], each of which generates a predefined aspect of the generated source code. Such a modular design also demands for an integration mechanism of each of these modules into the overall code generation process in order to generate the overall software system.

PR-2-2 Code generation adaptability: We consider adaptability of code generators as the ability to adapt a code generator to certain end user requirements that affect generated source code. Rather than directly manipulating a code generator, a black-box adaptation of transformations and templates is targeted, i.e., the code generator provides predefined spots for adaptation (in this thesis they are called *hook points* as explained in more detail in Section 8.3.1), and supports extension and adaptation of templates and transformations.

PR-2-3 Flexible use of transformations and templates: Especially for data-centric applications, a code generation approach influences the code generator's maintenance and development, because code generation of a GUIs differs from the code generation of an application layer (cf. [MK09]). In particular, for the GUI generation, a template-based approach is suitable because generated source code consists of non-changing source code blocks with minimal model-dependent parts. Code generation for an application layer benefits from a transformation-based approach, since the generated source code depends on the input model solely. Hence, an integrated approach has to be non-restrictive and facilitate flexible use of transformations and templates.

PR-2-4 Target language independent transformations: To support reuse of transformations, the proposed approach is restricted to object-oriented target languages and uses transformations on an CD4Code-AST. It describes the object-oriented structure of the synthesized source code (as understood in [Eli94]), but neglects implementation details. Hence, transformations on the CD4Code-AST can be reused for different input and object-oriented target languages.

PR-2-5 Well-formedness checking: An essential requirement when generating source code is to provide means to partially verify the well-formedness of the generated source code. We aim for an AST-based approach to check well-formedness of only the object-oriented structure such as classes, methods, and variables. Implementation details are not considered to keep the CD4Code ML target language independent (cf. *PR-2-4*).

PR-2-6 Target language-specific templates: Transformations independent of the target language do not provide sufficient means to write target language-specific source code such as method implementations. Hence, to define such implementation concerns, target language-specific templates can be provided. Such templates are bound to particular CD4Code-AST nodes and are used in concert with transformations to synthesize source code.

PR-2-7 Target language-specific default templates: The integrated approach provides a set of default templates for a particular target language. A default template defines the mapping of a particular CD4Code-AST node type to a target language concept.

8.2 Integration of Transformation- and Template-based Code Generation

The integrated transformation- and template-based code generation approach developed in this thesis is based on a partitioning of the overall code generation (i.e., code generation

used in MC as described in Section 2.2.4) into three steps employing transformations, templates, and an IR. A similar approach is used in reverse engineering [BCDM14]. It employs transformations and code generation for model extraction rather than providing flexible and adaptable code generation. In addition, an approach using transformation- and template-based code generation has been proposed by the openArchitectureWare system [HVEK07], which is now part of the Xtext project [www15c]. It defines workflows that contain a transformation step before generating source code via templates. However, support for AST-node-specific templates is lacking, which restricts adaptability of the code generator. Geiger et. al. proposed an approach that transforms an input model to a token tree and, afterwards, to a search tree to, eventually, use template-based code generation [GSR05]. Nevertheless, the token and search tree are target language dependent and adaptation concerns are not explicitly addressed.

The approach used in this thesis is shown in Figure 8.1. Code generation starts after a model has been successfully processed and the AST has been built up. The first step (1 in this figure) supports endogenous transformations on the input model-AST, i.e., the AST of the input model, only. Templates are not required, because the input model-AST is not used for synthesizing the target language source code.

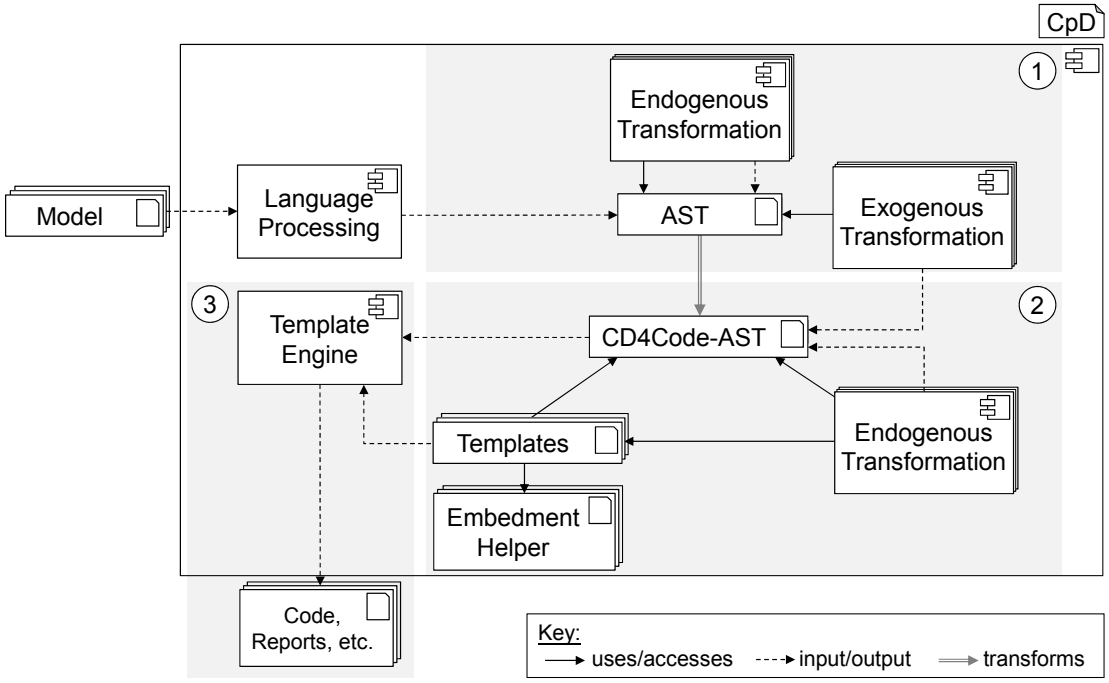


Figure 8.1: An overview of the integrated transformation- and template-based code generation, which is separated into three steps.

To continue to the second code generation step (transition between (1) and (2) in Figure 8.1), the input model-AST has to be transformed to the CD4Code-AST, which is an abstraction of object-oriented concepts, as explained in more detail in Section 4.3. Note that in MDA this transition is considered as PIM-to-PSM transition [Mel04].

In the second step ((2) in Figure 8.1), endogenous transformations on the CD4Code-AST are executed to consecutively transform it by manipulating higher-level object-oriented concepts such as classes, methods, an attributes. To realize target language implementation details, e.g., method implementation, templates can be attached to CD4Code-AST nodes (cf. *GR-2-3* and *GR-2-6*). These templates can be passed additional embedment helpers (cf. Section 2.2.4) or other CD4Code-AST nodes. There is no restriction when to use transformations or templates; or how to design them. Nevertheless, for effective use of this approach design guidelines are presented in Section 8.4.

Finally, in the last step ((3) in Figure 8.1), the transformed CD4Code-AST, the attached templates, and additional default templates for the target language are passed to a template engine and concrete source code is generated.

Subsequently, this code generation approach is described in detail using a case example.

8.2.1 Case Example: Statecharts-to-Java Source Code

Consider the following example, which aims to synthesize Java source code from a SC [www15b]. The input SC describes a simplified Ping-Pong game as shown in Figure 8.2. It consists of a `Ping` and a `Pong` state. Starting from the initial `Ping` state, the `Pong` state can be accessed when the stimulus `returnBall` is given. From the `Pong` state, the `Ping` state can be accessed by the same stimulus.

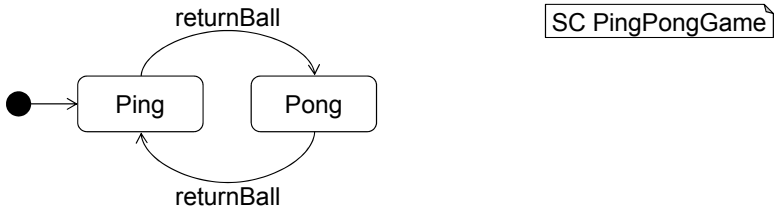


Figure 8.2: A SC for a simplified Ping-Pong game.

To synthesize executable Java source code for the Ping-Pong game, the State-Pattern [GHJV95] is used as shown in Listing 8.1. Note that for presentational reasons the implementation has been simplified. This pattern introduces a `Ping` (ll.27-31) and a `Pong` (ll.33-37) class for the equally named states. In addition, the `PingPong` class (ll.1-21) is added to manage the transitions.

In the remainder of this section, the different aspects of the proposed code generation approach are explained in more detail including the IR, model-to-model, and model-to-text transformations using this example.

```java
public class PingPong {
  private PingPongState state;
  private Ping initialState = new Ping();
  private Pong pongState = new Pong();

  public void returnBall() {
    state.handleReturnBall(this);
  }

  public void setState(PingPongState state) {
    this.state = state;
  }

  public Ping getPing() {
    return this.initialState;
  }

  public Pong getPong() {
    return this.pongState;
  }
}

public abstract class PingPongState {
  public abstract void handleReturnBall(PingPong p);
}

public class Ping extends PingPongState {
  public void handleReturnBall(PingPong p) {
    p.setState(p.getPong());
  }
}

public class Pong extends PingPongState {
  public void handleReturnBall(PingPong p) {
    p.setState(p.getPing());
  }
}
```

Listing 8.1: State-Pattern realizing the SC in Figure 8.2

8.2.2 An Object-Oriented Intermediate Representation using CD4Code

The CD4Code ML is the foundation of the presented code generation approach. Such an abstract representation of the generated source code can be used to validate well-

formedness by using context conditions for CD4Code (cf. *GR-2-5*). It also facilitates input and target language independence. Likewise, it improves reuse of code generation because endogenous transformations on the CD4Code-AST can be reused.

In general, similar approaches to use an IR for code generation have been proposed. An XML-based IR is used in the Clearwater code generation approach [SPJ+05]. Although the IR is not restricted to an input language or a target language, because it accepts arbitrary new tags, this IR hampers well-formedness checking. In [Rei15], an IR ("*Zwischenstruktur*") is created and used for template-based code generation to separate the input model and the target source code from the code generation process to achieve flexibility. Nevertheless, the IR is specifically restricted to UML CD input models and mixes AST and ST related information. Moreover, a meta-model transformation-based approach has been proposed in [HKGV10]. It uses the target language's meta-model as an IR. It has, additionally, been extended in order to allow merging. Another approach based on a Java IR with additional EJB extensions has been proposed to generate Java EJB applications [EBBG12]. The last two approaches use target language meta-models, which makes them target language dependent. Moreover, such a target language meta-model approach contains every implementation detail in the AST during code generation and is used by pure transformation-based code generation approaches (cf. [HKGV10]).

In this thesis, the goal is to synthesize object-oriented source code only. Hence, the IR is restricted to the general elements of object-oriented programming languages (cf. Section 8.1). This design decision is manifested in the design of the CD4Code ML, which is described in Section 4.3.

To demonstrate the use of CD4Code as an IR, the targeted Java source code's object-oriented structure of the case example in Section 8.2.1 is shown in Listing 8.2. In particular, the shown CD4Code model represents the object-oriented structure of the targeted Java implementation shown in Listing 8.1 on page 150. It starts with a model definition of the Ping-Pong game (l.1) and contains each of the Java classes to be generated (ll.2-11, ll.13-15, and ll.17-23). Each of them has public visibility defined and contain all variables and method declarations but omits method implementations, e.g., ll.7-10.

While the chosen case example is kept simple for presentational reasons, the mapping of the input model-AST to the CD4Code-AST requires knowledge of the input and target language as well as the code generator itself and, hence, has to be done by the generator developer (cf. Section 3.1.3). The development of such exogenous transformation can be supported by domain-specific transformation languages (e.g., [Wei12]). Another approach, which is also used in this thesis, is to provide sophisticated APIs to create and manipulate the CD4Code-AST, as described in Section 9.2.

8.2.3 Model-to-Model Transformations

The proposed approach also relies on model-to-model transformations - endogenous and exogenous transformations described in Section 2.2.4 - to manipulate the input model-

CD4Code

```
classdiagram PingPongGame {
  public class PingPong {
    private PingPongState state;
    private Ping initialState = new Ping();
    private Pong pongState = new Pong();

    public void returnBall();
    public void setState(PingPongState state);
    public Ping getPing();
    public Pong getPong();
  }

  public abstract class PingPongState {
    public abstract void handleReturnBall(PingPong p);
  }

  public class Ping extends PingPongState {
    public void handleReturnBall(PingPong p);
  }

  public class Pong extends PingPongState {
    public void handleReturnBall(PingPong p);
  }
}
```

Listing 8.2: An example of the IR of the example in Listing 8.1 described using the CD4Code ML.

AST, the CD4Code-AST, and transfer the input model-AST into an CD4Code-AST. Each type of transformation is understood as defined in Def. 8 on page 17.

Two types of endogenous transformations are used. The first type are endogenous transformations on the input model-AST (step (1) in Figure 8.1 on page 148). Such endogenous transformations on the input model-AST are independent of CD4Code and the target language and are required if:

- Not every input language concept can clearly be mapped to concepts of CD4Code but can be resolved by "desugaring" [KV10] or normalization (cf. Section 2.2.4).
- A simplification of the mapping of the input model-AST to a CD4Code-AST in terms of amount of classes, methods, etc is wanted.

For example, if the input model is a SC, as in the example in Section 8.2, hierarchies in the SC can be flattened.

The second type of transformations are endogenous transformations on the CD4Code-AST. They are used to modify the implementation model that is used for code generation. For instance, a `serialize()`-method can be added to the CD4Code-AST nodes of the Ping-Pong game (shown in Listing 8.2 on page 152) to realize serialization. Every modification done by such transformation directly influences the generated source code. Such endogenous transformations on the CD4Code-AST support modularity of the code generator (cf. *GR-1-1*), because they can be arbitrarily combined and used, and favor reuse, because they are independent of the input model and target language (cf. *GR-2-4*).

In addition to endogenous transformations, exogenous transformations are used to transfer the input model-AST to the CD4Code-AST (transition between (1) and (2) in Figure 8.1). Exogenous transformation can either

(i) built up the complete CD4Code-AST from the input model-AST or

(ii) can enrich an already existing CD4Code-AST, which has previously been created by parsing a valid CD4Code model, with input model-specific parts.

The primer approach, i.e., (i), is suitable when the CD4Code-AST can solely be build from the input model and does not require any additional static elements, e.g., technical classes. In contrast, using a predefined CD4Code model, i.e., (ii), is suitable when the object-oriented structure is mainly static or contains additional elements, which are required for the generated source code but not defined in the input model.

This integrated code generation approach can be applied to arbitrary input languages. However, exogenous transformations from the input model to the CD4Code-AST require a clear mapping of the input language concepts to object-oriented concepts. For generating data-centric applications from CD4A models, the exogenous transformations are defined by the mapping in Chapter 5 and in Chapter 7. For other input languages this mapping may not make sense, is ambiguous, or infeasible.

For example, a sequence of transformations may still produce syntactic invalid source code, because of CD4Code-AST nodes required by one of the transformation rule (Def. 8) cannot be found or the synthesized source code has a static dependency to some other source code produced by another transformation, which has not been executed. An approach to address this issue is to introduce preconditions for transformations, e.g., [SBM09, RBP+14, MSCB15]. However, such preconditions are understood as guards that have to be true in order to execute a transformation but do not consider static source code dependencies on generated source code level. An alternative approach is to introduce additional target language constructs, e.g., partial classes, and context information, to enable modularity of transformations [HKGV10]. Or frameworks for safe composition of transformations [HKA10]. Finally, large scale transformation chain management [BPdOB13] addresses transformation variability and explicit transformation dependencies (on transformation and generated artifact level). However, this approach still has limitations, e.g., co-evolution, and is not applicable in an agile environment (cf. [BPdODF14]).

In this thesis, a methodological approach is proposed to design reusable transformations and avoid such issues, as explained in more detail in Section 8.4.1.

8.2.4 Adding Implementation Details via Template Attachments

In the integrated approach, the template-based part (③ in Figure 8.1) is used to realize a model-to-text transformation (*model-to-code* transformation in [SVC06]) by synthesizing source code from the CD4Code-AST. Therefore, this transformation uses default templates, which specify the mapping for every CD4Code language concept, i.e., CD4Code-AST type specific templates, to one particular default target language concept (cf. *GR-2-7*). For example, mapping methods to methods with an empty method body. To add implementation details such as method implementations to one or multiple CD4Code-AST nodes, default templates have to be replaced. Hence, we define a *template attachment* as follows:

Definition 20 (Template Attachment)**.** *A template attachment is a replacement of a default template for a particular CD4Code-AST node by a template.*

All template attachments have to be attached before entering ③ of the code generation process by endogenous transformations on the CD4Code-AST in step ②. Each CD4Code-AST node can be attached multiple CD4Code-AST-specific templates. By default, each template attachment is a default template for the CD4Code-AST node's type (from the set of default templates). If no default template is defined for this CD4Code-AST type, no template attachment exists.

Template attachments are handled for each AST node individually and can only be defined once for an CD4Code-AST node. Hence, endogenous transformations on the AST node do not affect template attachment as long as the AST node is not deleted or copied. In this case, the template attachment is deleted as well.

To demonstrate template attachments, the example in Figure 8.3 shows an excerpt of the CD4Code-AST from Listing 8.2. Note that to create the CD4Code-AST from the CD4Code grammar, the approach described in Section 2.2.2 has been applied. In this example, templates are attached to implement method bodies such as the `r:ASTCDMethod`, which delegates the stimulus to a particular state, or the `h2:ASTCDMethod` to set the new state `Pong`. Default templates are neglected for presentational reasons.

In general, different approaches integrating transformation- and template-based code generation have been proposed. In particular, an approach to integrate template-based code generation into graphical model transformations has been proposed in [Gir08]. It enables graphical modeling of endogenous transformations and define method body strings via templates. A similar approach to use templates for implementation concerns has also been presented in [BV06]. In contrast, our proposed approach employs endogenous and exogenous transformations. In addition, templates can also be used to define, e.g., default variable values, or even complete classes and, hence, are not restricted to

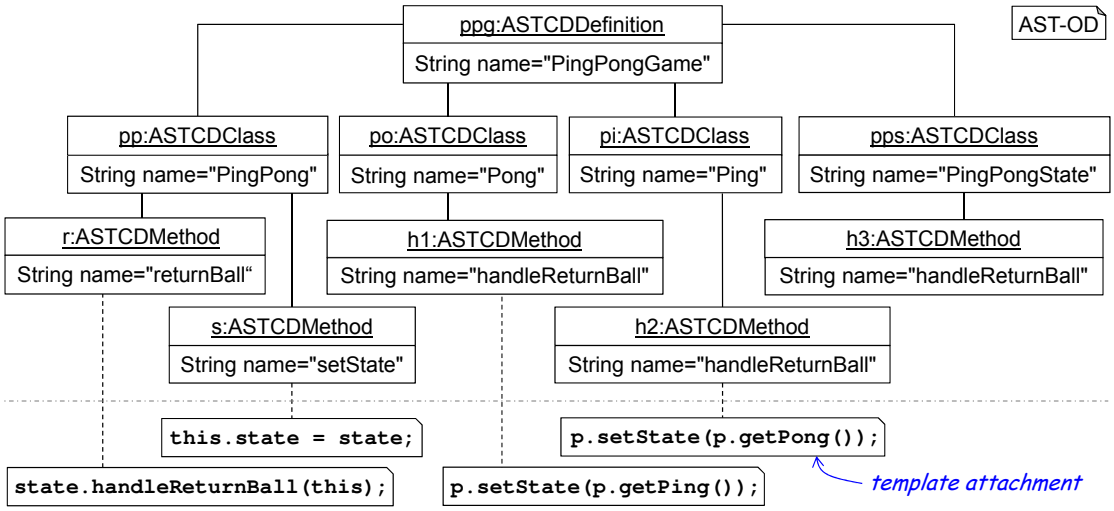


Figure 8.3: An excerpt of the CodeCD-AST for the Ping-Pong game shown in Listing 8.2 with additional templates attached to implement method bodies.

method bodies only. Moreover, [GSR05] allows to attach a template to the root node of the search tree to handle all child nodes. Conversely, our approach allows to define template attachments for arbitrary CD4Code-AST nodes.

8.2.5 Model-To-Text Transformation

The actual model-to-text transformation to produce source code starts by passing the transformed CD4Code-AST, the template attachments, and a set of default templates to the template engine ((3) in Figure 8.1). The responsibilities of this step are:

(i) Traversal of the CD4Code-AST.

(ii) Execution of templates in the template attachment.

(iii) Writing of output into a file.

To demonstrate the code generation, consider the example shown in Figure 8.4. We assume that the set of default templates contains templates for CD4Code-AST nodes. In this particular case, the `Class.ftl` template is defined for the `ASTCDClass` type, the `Method.ftl` template is defined for the `ASTCDMethod` type, and the `Attribute.ftl` template is defined for the `ASTCDAttribute` type. The `ASTCDDefinition` type does not have any default template. Additionally, CD4Code-AST-node-specific templates are defined for `c:ASTCDAttribute` (`IntValue.ftl`), `d:ASTCDMethod` (`ToString.ftl`), and `b:ASTCDClass` (`MyClass.ftl`).

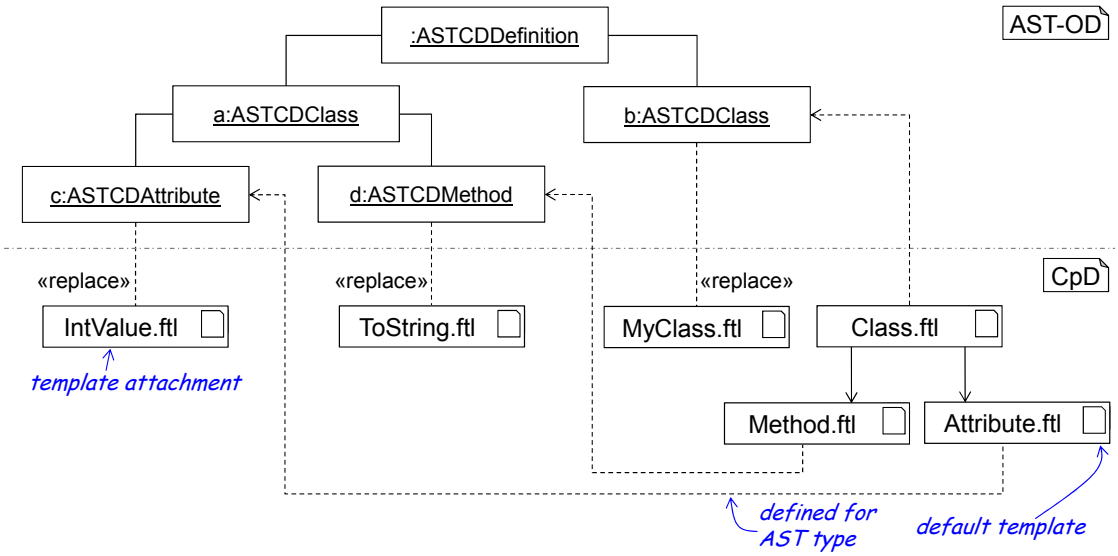


Figure 8.4: An CD4Code-AST with template attachments and a set of default templates. The execution order of all attached templates and default templates is computed during template engine execution.

When the template engine is executed for this example, it traverses the CD4Code-AST in a depth-first way. Thereby, the following templates are executed in the order from top to bottom:

1. For `:ASTCDDefinition`, no template is called, because none is defined.
2. For `a:ASTCDClass`, the default template `Class.ftl` is called.
3. For `c:ASTCDAttribute`, the template `IntValue.ftl` is called instead of the default template `Attribute.ftl`.
4. For `d:ASTCDMethod`, the template `ToString.ftl` is called instead of the default template `Method.ftl`
5. For `b:ASTCDClass`, the `MyClass.ftl` template is called.

In order to write source code into a file, it has to be specifically defined for which AST type a file is created. In this example, we define that for each CD4Code class the source code resulting from traversal of the spanned subtree is written into a file. Hence, for the `a:ASTCDClass` and `b:ASTCDClass` instance two separate files containing the evaluated template content's result are created.

This example assumes Java as the target language. It is possible to reuse the transformations to generate different target language. However, in this case the set of default templates has to be exchanged because the default templates are responsible to map CD4Code language elements to the target language. For example, abstract CD4Code classes is mapped to abstract Java classes but for C++ abstract classes have to be mapped to virtual methods. Moreover, the CD4Code-AST-node-specific templates have to be exchanged as well, because they are target language agnostic.

8.3 Template Adaptation via Template Hook Points and Template Extensions

Synergetic transformation- and template-based code generation aims for black-box adaptation of the code generation process. Hence, in the remainder of this section, template hook points and template extensions are presented to realize adaptation of the code generation process if the template architecture is known.

8.3.1 Adaptation via Template Hook Points

Template attachments are primarily targeted to replace default templates and provide implementation details for individual CD4Code-AST nodes. In some cases, this is not practical. For example, if existing default templates are reused but have to be adapted in minor parts.

An approach to adapt parts of code generator templates in a predefined way are *template hook points*. This approach is based on the idea of variation points [CN01, PBvdL05] and hooks methods [Pre95]. Each template hook point allows to adapt a template in a dedicated place without manually changing it (cf. *GR-2-1*). A template hook point is defined as follows:

Definition 21 (Template Hook Point)**.** *A template hook point is a spot in a template planned for customization.*

Typically, template hook points are designed during code generator development and assigned a unique name. Note that template hook points can be added afterwards if needed, as proposed in the method in Section 3.3.2. In contrast to hot spots, template hook points affect multiple artifacts, i.e., adapting a template hook point affects all artifacts that are generated by this template.

Adapting a template hook point means to set one of the following values to override the default value, which may also be empty:

(i) **Fixed String**: A hook point's value is set to a plain `String` value.

(ii) **Template Name**: Instead of directly defining a `String`, a template is specified. This template is executed on the available AST nodes and the resulting value is used for the template hook point.

(iii) **Template Content**: A template string value is a `String` that contains FreeMarker expressions. It is processed by the FreeMarker template engine and the result is set for the hook point.

(iv) **Executable Code**: The value of a template hook point is defined as the return value of a Java method, which is typically a String value. The value is inserted into the template hook point.

Each provided hook point is set by (senior) application developers for extension and adaptation purposes. In general, the aforementioned values for adapting hook points have distinct benefits and drawbacks. In particular, (i) and (iii) are suitable, when the adaptation can be expressed by a simple string or a simple template expression. This may become a drawback in the maintenance of the code generator, if the used values are large strings or template expressions because tooling cannot be used. Furthermore, when using (i) and (iii) for adaptation, transitive adaptation, i.e., adapt the adaptation using hook point values, are not feasible. Hence, approach (ii) overcomes this limitation by allowing the use of manually-written templates. However, while the template content may be arbitrary, only the AST-nodes passed to the hook point can be used. Finally, (iv) provides the most powerful adaptation approach, because it allows to use arbitrary executable source code. However, it requires recompilation of the code generator such that the source code can be executed. This is not necessary for templates, because they are interpreted.

8.3.2 Adaptation via Template Extensions

Besides adapting templates via template hook points, code generation can directly be customized using *template extensions*, if the internal template architecture is known. In this thesis, a template extension is understood as follows:

Definition 22 (Template Extension)**.** *A template extension is an assignment of a list of templates to a particular CD4Code-AST type.*

Template extensions are evaluated for individual CD4Code-AST nodes and contain template attachments. The following modifications for template extensions are provided:

- **Replace Operation:** Replace a particular template from the template extension.
- **Add-Before Operation:** Prepend a template to the template extension.
- **Add-After Operation:** Append a template at the end of the template extension.

Template extensions and operations on them provide a powerful approach that, however, may result in conflicting operations. In the remainder of this section, all conflicts and their resolution are explained in detail based on the overview shown in Figure 8.5.

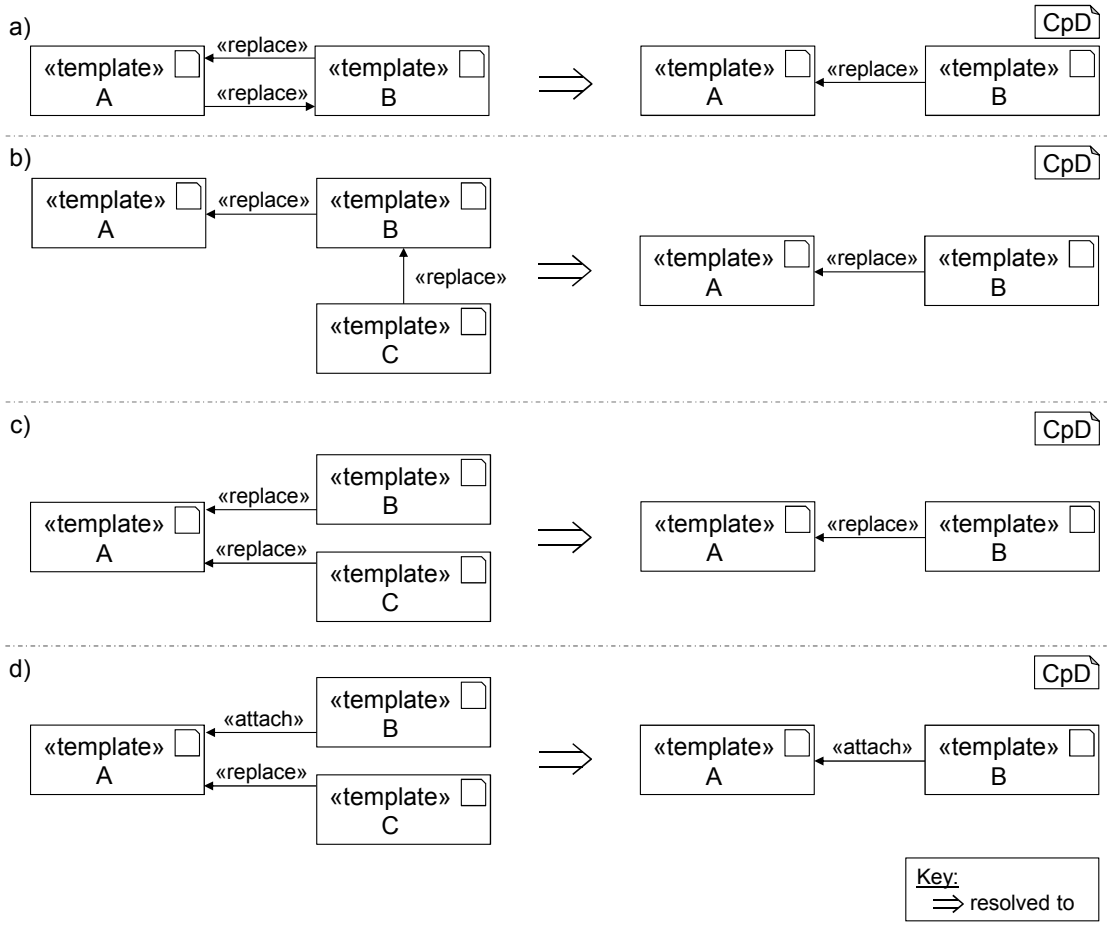


Figure 8.5: Resolution of conflicting template extension operations is done by only executing the first replace operation (a) and (c) in a non-transitive manner (b); and prioritize conflicting template attachments and template extensions (d).

To avoid cyclic add or replace operations, i.e., (a) and (b) in Figure 8.5, resolving template extensions is not performed transitively but for only one operation. All further operations are omitted. For example, in the example (a) the first operation (execution order is from top to bottom) is the replacement of template A with template B. Hence, the result is only the execution of first operation only. Similar holds for example (b) (execution order is from left to right and top to bottom), where only the replacement of template A with template B is executed.

However, because operations on template attachments are executed sequentially, run-time errors may occur. For example, when two *replace operations* sequentially replace the same template, i.e., (c) in Figure 8.5, the latter replace operation will fail, because the template has previously been replaced. In aspect-oriented programming this is considered as the *"fragile point cut"* problem [SK04]. An approach to detect such errors is to process the sequence of template attachment operations a priori and analyze them to detect conflicts. However, this will involve processing the complete code generator's source code. Alternatively, another approach is to detect such conflicts at run-time of the code generator and inform the generator developer but do not stop code generation. In this thesis, the latter approach has been chosen to prevent template processing.

It is also possible that a template attachment conflicts with a replace operation, i.e., a template attachment exists that is not the default template and the default template is to be replaced by a template extension. For example, (d) in Figure 8.5. In this case, the template attachment is executed and the replacing operation is ignored. This convention is used to resolve similar conflicts.

Note that if a template from a template extension is replaced, all before and after templates of the replacing template are added.

Operations on template extensions can introduce syntax errors in the generated source code. For example, it is possible to use incomplete target language statements in templates, which combined with other templates produce syntactically correct source code. However, if these templates are executed in the wrong order or are replaced, the resulting source code is syntactically incorrect. Although it is possible to restrict templates to produce syntactical correct target language statements only (as shown in [ZR11a, ZR11b]), it is not practical because it restricts the overall code generation approach, i.e., templates can then only be replaced by syntactically correct templates and multiple attached templates are not possible. However, because the presented approach is not restrictive and syntax errors may occur, we provide methods to avoid such pitfalls (cf. Section 8.4).

8.3.3 Technical Realization in MontiCore

The MC framework uses the `TemplateController` class to manage template execution, which allows to define hook points, as shown in Figure 8.6. The `TemplateController` contains a `defineHookPoint(String n)`-method to define a hook point. If the hook point has already been defined and, hence, the name is not unique, a warning is raised during code generator execution. To avoid such name conflicts, a naming convention can be used during code generator development, e.g., by using the template name as a prefix.

The defined hook points are managed by the `GlobalExtensionManagement` class. It contains the `bindHookPoint(String hookName, HookPoint hp)`-method to bind a value to an existing hook point. Moreover, the `existsHookPoint(String hookName)`-method can be used to check if a hook point does already exist. Each hook

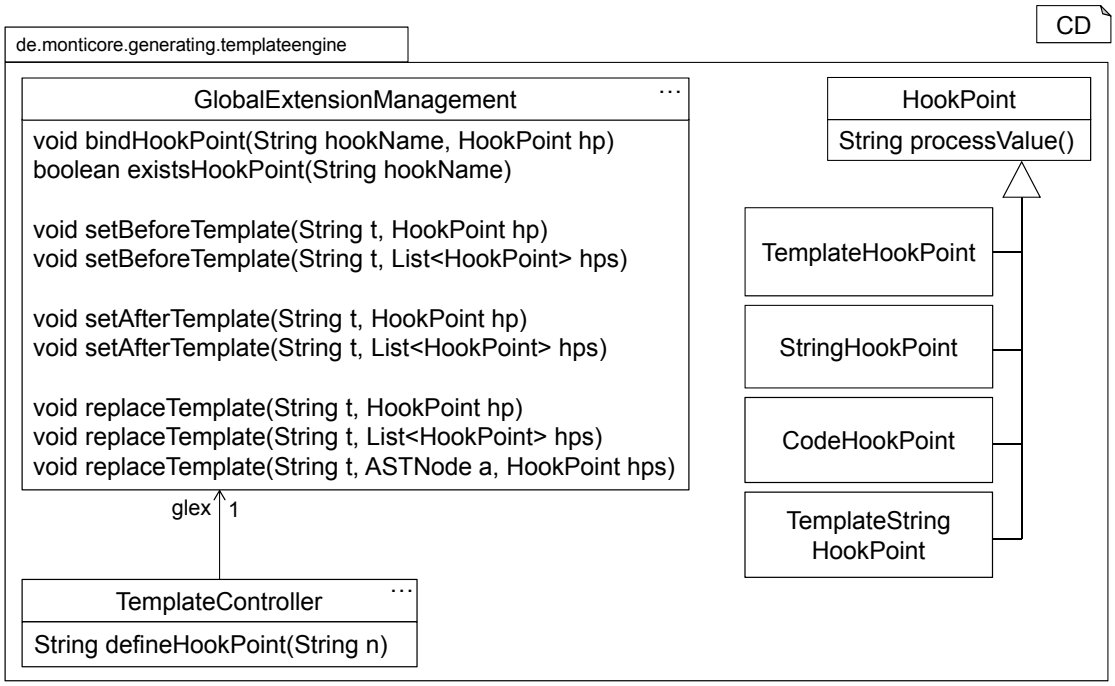


Figure 8.6: The technical realization of template hook points and template extensions.

point can be assigned a template, string, code, or template string value (cf Section 8.3.1). Each value is realized as a subclass of the `HookPoint` class, as shown Figure 8.6.

In addition, the `GlobalExtensionManagement` class offers methods for each template extension operation, i.e, set before, set after, and replace. All methods are shown in Figure 8.6. Technically, template extensions are also `HookPoints`. Hence, not only templates can replace other template but also `StringHookPoint`, `TemplateString-HookPoing` and `CodeHookPoint` values. For example, the `setBeforeTemplate( String t, HookPoint hp)`-method allows to add the hook point value `hp` before all others for the hook point with the name `t`. All hook point types can be used for template extensions. Similar holds for template attachments, i.e., a template attachment is set using the `replaceTemplate(String t, ASTNode a, HookPoint hps)`-method.

For example, Listing 8.3 gives an example of using the provided API. Before templates can be replaced or hook points can be attached, an instance of the `GlobalExtension-mangement` and the `TemplateController` has to be created. This is done by setting the template engine (ll.3-4 and l.9) and a `FileReaderWriter` to write output files (l.6 and l.10). In addition, a path for external templates can be set (l.12), the output directory has to be set (l.13), and tracing can be enabled (l.14). With this configuration, an instance of the `TemplateController` can be retrieved.

To replace the existing template `EmptyMethod`, the `replaceTemplate` method is called (l.20) and an instance of the `TemplateHookPoint`, which realizes the template name hook point (cf. Section 8.3.1), is passed (l.21). In addition, in l.23 an already defined hook point is bound to the string `"@Entity"` (l.24).

```
1 GlobalExtensionManagement glex =                                    Java
2     new GlobalExtensionManagement();
3 FreeMarkerTemplateEngine ftl = new FreeMarkerTemplateEngine(
4     new FreeMarkerConfigurationBuilder().build());
5
6 FileReaderWriter fh = new FileReaderWriter();
7 TemplateControllerConfiguration cfg =
8   new TemplateControllerConfigurationBuilder().glex(glex)
9     .freeMarkerTemplateEngine(ftl)
10    .fileHandler(fh)
11    .classLoader(getClass().getClassLoader())
12    .externalTemplatePaths(new File[]{})
13    .outputDirectory("out")
14    .tracing(true)
15    .build();
16
17 tc = new TemplateController(cfg, "");
18
19 // replace existing template
20 glex.replaceTemplate("EmptyMethod",
21                      new TemplateHookPoint("InitMethodBody"))
22 // bind a string to an existing hook point
23 glex.bindHookPoint("Class::addAnnotation",
24                    new StringHookPoint("@Entity"));
```

Listing 8.3: An example of using the provided API to replace templates (ll.20-21) and bind hook points (ll.23-24).

8.4 Methods for Transformation Design and Management

For effective use of the proposed code generation approach methods and guidelines for transformation design and management are presented in the remainder of this section.

8.4.1 Method for Transformation and Template Development

The provided flexibility to switch between transformation- and template-based code generation in step (2) of the code generation approach (Figure 8.1 on page 148) can produce unwanted side-effects such as unmaintainable templates or transformations and

an inflexible transformation sequence. In consequence, a method and guidelines to guide generator developers to reduce these disadvantages are needed.

The method and guidelines used in this thesis (i) are based on the assumption that an initial exogenous transformation transformed the input model to the CD4Code-AST, (ii) neglect testing concerns, because it is an essential part during development and can be done as described in [KLM+16], and (iii) require the existence of a reference implementation (cf. [Sch12]), which is a manually-written implementation of the targeted software system's functionality for one particular CD4A model.

The proposed method is shown in Figure 8.7. Based on a reference implementation, in the first step, the overall source code is partitioned into a model-dependent part, which has to be derived from the model, and model-independent part, which remains static even if the model changes. The model-dependent part is to be generated by one or multiple transformation and templates.

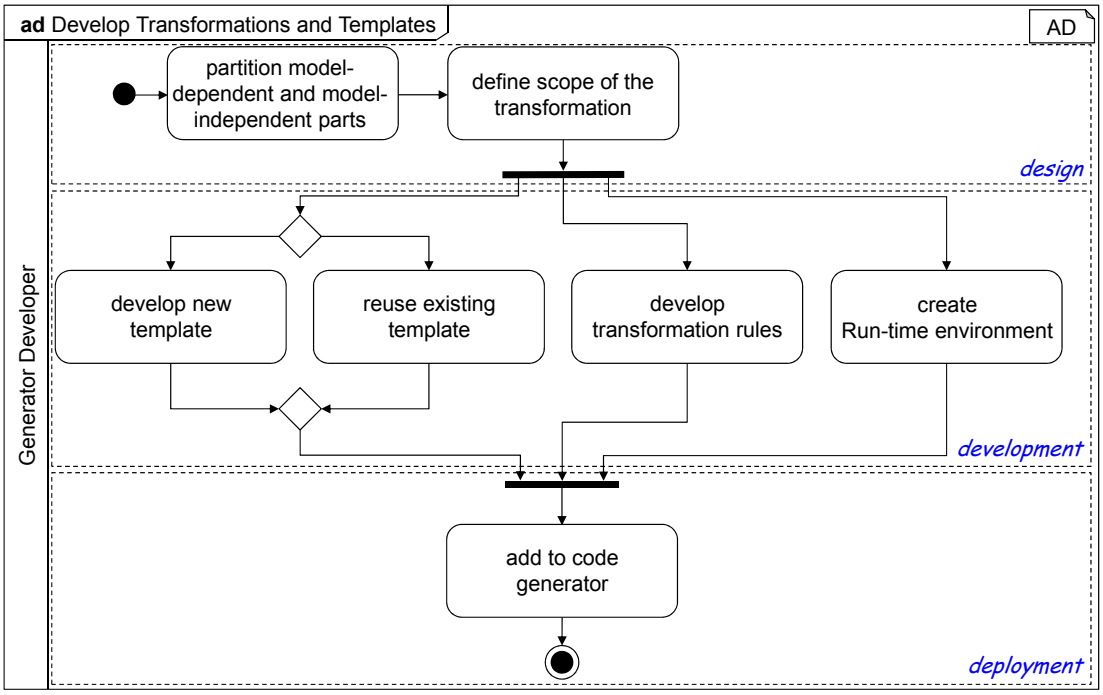


Figure 8.7: A method to develop code generator transformations and templates.

In the next step, the scope of the transformation under development is defined. It contains the artifacts and source code that should be generated. A guideline to identify the transformation's scope is to choose the parts of source code that form a functional complete working unit, i.e., source code that implements a particular functionality of the targeted software system and can be generated standalone. Whenever this is not possible

the main goals in defining the scope are high cohesion and low coupling of the generated source code. On the transformation level the goal should be to aim for *horizontal separation of concerns*, i.e., reduction of dependencies between transformations [HKGV10]. However, interconnections on the generated source code and transformation level have to be documented to make them explicit and support reuse. It is possible that in this step multiple transformations have been identified and need to be developed. If multiple transformations have been identified, the subsequent steps have to be performed for each transformation individually.

After the particular part of the source code to be generated by one transformation has been identified, the model-independent part of this particular piece of source code is placed in a RTE (cf. Section 7.1.3). For the model-dependent part, transformation rules are created that map input model concepts to the CD4Code-AST. All parts of the source code to be generated that cannot be mapped to the CD4Code-AST because the CD4Code does not support it, e.g., method bodies, annotations, and variable assignments, have to be handled by templates and template attachments.

Before developing a new template, the generator developer should check if there are already existing templates to be reused. This may also imply adapting existing templates by using further customization approaches, e.g., hook points (cf. Section 9.3). Afterwards, it is attached to an CD4Code-AST node.

Technically, there is a strong bond between a transformation and templates, because neither of them can be used standalone. Hence, this should be visible in the code generator by, e.g., placing them in the same package or folder. Likewise, the documentation has to contain this relation.

Finally, after the transformation and necessary templates have been created, they have to be added to the code generator. This may imply adapting default configurations of the code generator (as explained in Section 9.3), deploying the transformations and templates with the code generator, and providing a documentation on its usage.

Chapter 9

MontiDEx: MontiCore Data Explorer Code Generator

The CD4A-to-Java source code mapping (cf. Chapter 5) and the mapping of CD4A to a data-centric infrastructure (cf. Chapter 7) are designed such that they can be performed by code generators. However, for efficient and effective use of a code generator, the generator developer has to address the different needs of potential users (cf. Section 3.1.3). In particular, (i) a code generator has to be a black-box for application developers and has to fit the application developer's development environment. At the same time, (ii) a code generator has to be designed in a configurable and adaptable way such that it can be maintained and extended by generator developers and can be adapted by senior application developers (cf. *RE-3*). Therefore, the transformation- and template-based code generation approach (cf. Chapter 8) has been proposed. In addition, for MDP of data-centric applications, a code generator also (iii) has to be designed to generate data-centric application prototypes from underspecified models.

In this thesis, the *MontiCore Data Explorer (MontiDEx) Code Generator* (cf. *RE-7*), which is technically based on the MC code generation framework (cf. Section 2.2), has been developed to realize the developed concepts and mappings for data-centric applications. It processes CD4A and ADJava models to generate data-centric applications and data-centric application prototypes (subsequently called *MontiDEx products*). The MontiDEx code generator provides defaults to handle underspecification, and facilitates code generator reuse (cf. *RE-4*) and configuration by configuration scripts, which supports configuration of every step of language processing and code generation (cf. *GR-1-1*).

The goal of this chapter is to provide a method to implement an adaptable and configurable code generator. Hence, this chapter is structured as follows. First, developed defaults to handle underspecification in CD4A models are presented in Section 9.1. Afterwards, the MontiDEx code generator's architecture is described in Section 9.2. Next, approaches for code generator configuration and adaptation are presented in Section 9.3. To increase understandability and maintainability of the code generator, reports are introduced in Section 9.4. Finally, in Section 9.5, methods to use, customize, and extend the MontiDEx code generator are presented.

9.1 Technique to Handle Underspecification in MontiDEx

MontiDEx[1] generates for every well-formed CD4A model executable Java source code. Due to the nature of CD4A models, this implies dealing with underspecification, because it contains semantic variation points by design. This underspecification is, however, required to define well-known system properties but yet omit unspecified ones. If underspecification is prevented by a code generator, its use in early stages of MDD, where parts of the model are not specified yet, is hampered. Even worse, MDP is infeasible, because prototypes only describe certain aspects of the overall software system [Som10]. Interested readers are advised to refer to [Rum12] for advantages and disadvantages of underspecification in UML CD.

The MontiDEx code generator handles underspecification by using predefined but reasonable defaults (cf. *RE-5-2*). Such a *default* is a design decision to resolve a particular underspecification of the input model. It is automatically taken by the code generator at its run-time and designed by the generator developer. Technically, a default can be realized by:

(a) Enriching the input model using endogenous transformations on it ((1) in Figure 8.1 on page 148).

(b) Exogenous transformations can be used to map underspecified concepts to predefined target language concepts in the IR ((2) in Figure 8.1 on page 148).

(c) Endogenous transformations on the IR assign defaults.

However, the first approach, i.e., (a), distorts the input model and may cause unwanted side-effects during code generation. The second approach, i.e., (b), cannot handle all underspecification, because the final software system is unknown at this point in time. Finally, the latter, i.e., (c), requires that every endogenous transformations on the IR handles underspecification independently.

In this thesis, (a) and (c) are used in concert. In particular, we propose to use (a) whenever underspecification can be resolved globally for the overall code generation process, e.g., a missing cardinality, and to use (c) to handle certain underspecification locally for individual parts of the code generation process, e.g., the number of keys in a qualified association.

A code generator that performs such modifications based on predefined defaults to resolve underspecification takes these always relative to the targeted domain and software system. The decision to take a default may be rooted on, e.g., structural properties of the input model such as class hierarchies, but also on existence of external artifacts. Hence, such defaults and the behavior of a code generator do not always fit. Even for

[1] MontiDEx is available at `www.monticore.de/dex`.

the same domain but with different requirements, such defaults may not be well suited. To address this issue, code generators allow for configuration and adaptation in order to override such defaults, which is explained in more detail in Section 9.3.

In the remainder, the MontiDEx code generator's defaults for underspecification in the CD4A input model are described. Further defaults and assumptions of the CD4A ML and the mapping are mentioned in Chapter 4, Chapter 5, and Chapter 7.

9.1.1 CD4A Underspecification and Defaults

An underspecification of the CD4A ML is the absence of cardinalities for associations. If a cardinality is omitted, the default cardinality `[*]` is used. However, if the association is qualified, the cardinality is `[1]`. Moreover, in case the underspecified association is a composition, the default cardinality for the right composition end (part) is `[1]`.

Another underspecification of associations is the navigation direction `--`. The default navigation in this case is a bidirectional navigation `<->`, which is a similar assumption as in [SBPM09]. However, if one of the association ends is an external type or a CD4A enumeration, the navigation from this association end is prohibited (cf. Section 6.4.1).

An overview of all cardinality and navigation defaults is shown in Table 9.1, where `A` and `B` are classes defined in the model and `E` is an external class.

Underspecified Association	**MontiDEx default**
`association A -> B`	`[*] A -> B [*]`
`association A <-> B`	`[*] A <-> B [*]`
`association A -- B`	`[*] A <-> B [*]`
`association A [String] -> B`	`[*] A [String] -> B [1]`
`association A -- E`	`[*] A -> E [*]`
`composition A -- B`	`[1] A <-> B [*]`
`composition A -- E`	`[1] A -> E [*]`

Table 9.1: Cardinality and navigation underspecification in the CD4A ML and the defined defaults (`E` is an externally defined type).

Another underspecification of the CD4A ML is the qualified association key size, which groups a set of association links. The number of links is specified by the qualified association's cardinality. However, there is no means to specify the number of keys (cf. Section 5.1). Hence, the MontiDEx code generator assumes in such cases that an arbitrary number of keys can exist.

9.2 MontiDEx Architecture and Technical Realization

As MontiDEx is a MC generator, its internal architecture is an extended variant of the architecture introduced in Section 2.2 as illustrated in Figure 9.1. It is extended with a set of predefined configurations, which predefine different variants of the generated product, as proposed in Section 7.1.3. Moreover, the code generation approach proposed in Chapter 8 is used. A core extension are *MontiDEx modules*, each of which encapsulates one transformation, a set of templates, and a set of embedment helpers. Each transformation uses the input model's AST or the output AST but only manipulates one of them to realize endogenous and exogenous transformations. A methodological approach to design such modules is presented in Section 8.4. In addition, the overall architecture is designed to allow adaptations and extensions of transformations and templates.

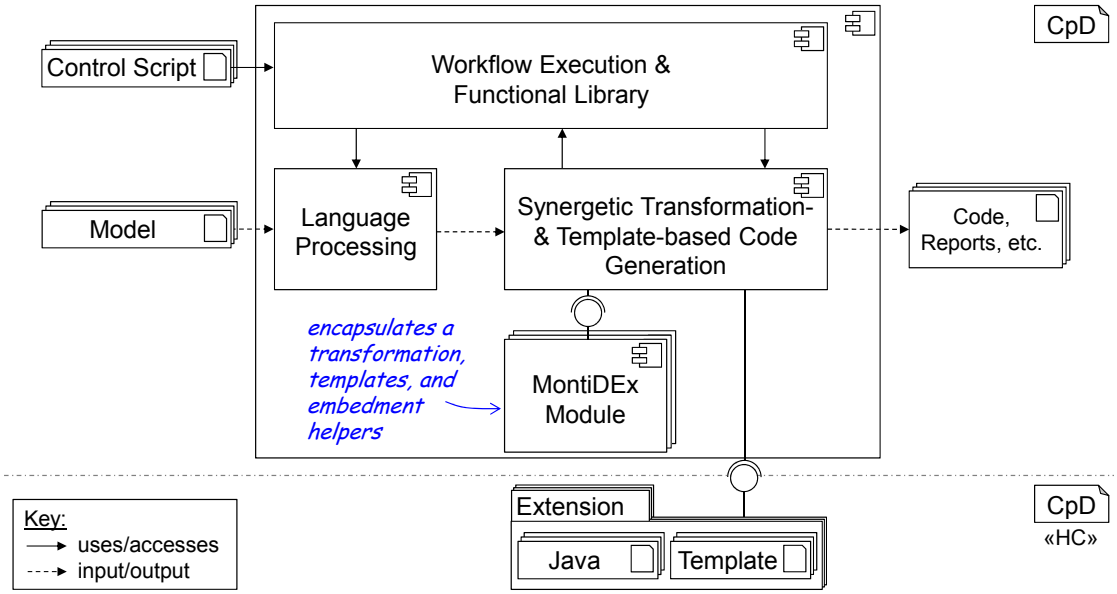


Figure 9.1: An overview of the MontiDEx code generator architecture.

In the remainder of this section, we present the infrastructure provided by the MontiDEx code generator to support generator developers in creating MontiDEx modules. Afterwards, we present methods and concepts to realize adaptation of a code generator in Section 9.3.

9.2.1 Technical Realization of the Common Infrastructure

Technically, the MontiDEx code generator is structured into six packages: `common`, `configure`, `reporting`, `cd2data`, `cd2swing`, and `cd2persistence`. Each

package serves a designated purpose, e.g., the `cd2data`, the `cd2swing`, and the `cd2-persistence` realize the mapping described in Chapter 5, and Chapter 7.

The `common` package contains shared embedment helpers to support creation of MontiDEx modules. It has a `builder` subpackage that contains extended MC builders to ease the creation of CD4Code-AST nodes by predefining default values, e.g., default visibility. Figure 9.2 illustrates the package's content and structure. Note that for presentational reasons we omit an explicit description of each builder, because its name indicates the AST node it creates. Furthermore, the `ConcreteModifierDelegate` and the `ModifierModifiable` handle modifiers and are omitted as well.

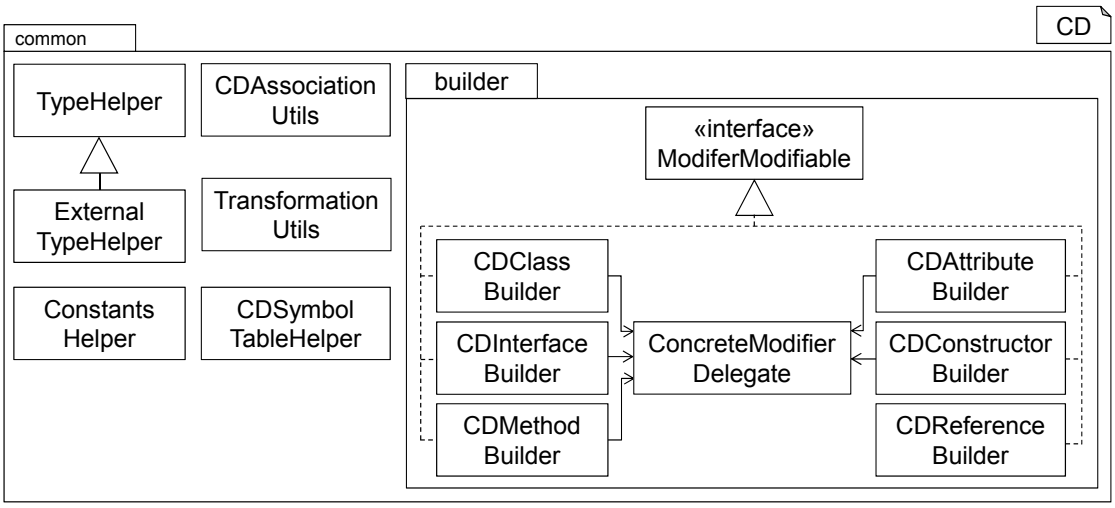


Figure 9.2: An overview of the `common` package, which groups common templates, embedment helpers, and CD4Code-AST node builders.

In the following, each remaining class in this package is explained in detail from a technical perspective.

AbstractTypeHelper and TypeHelper Embedment Helper

The `AbstractTypeHelper` and `TypeHelper` are embedement helpers that are used to check for a particular type of an CD4Code-AST node, e.g., for checking whether an attribute is of type `String`. It supports primitive, wrapper, collection, and external types. Additionally, methods are provided to convert primitive data types into wrapper types and methods to get a default value for a particular type, as shown in Figure 9.3.

In the remainder, the most relevant methods are explained in more detail.

- **`getDefaultValue(ASTType s)`**: For each CD4Code-AST primitive type (`int`, `boolean`, `short`, `byte`, `float`, `long`, `double`, and `char`), a default value is

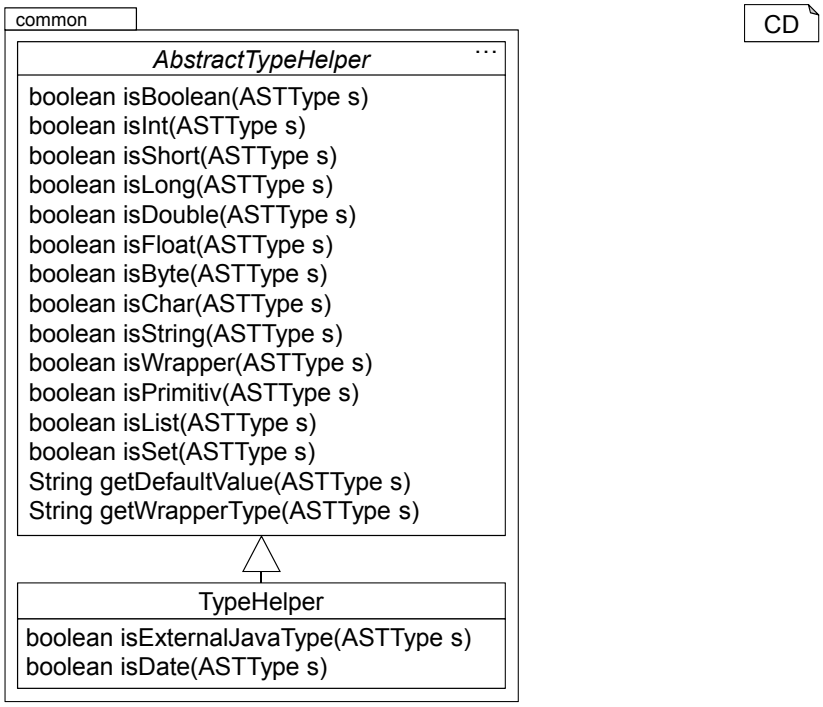


Figure 9.3: Methods provided by the `AbstractTypeHelper` and the `TypeHelper` embedment helper.

returned. For wrapper types and `String`, an empty default value is assumed. External types are not supported. Hence, the default value in this case is `null`.

- **`getWrapperType(ASTType s)`**: A primitive wrapper type of a primitive data type is a class encapsulating the primitive data type (cf. Chapter 4). This method provides a primitive wrapper type for a given primitive type. If no wrapper can be found `String` is used.
- **`isExternalType(ASTType s)`**: An external type is a type not defined in the CD4A input model (cf. Section 6.4.1). This method checks if the given type is an external data type.

CDAssociationUtil Embedment Helper

This embedment helper groups methods for accessing properties of CD4Code associations, e.g., checking if the association is qualified in both directions. Its content is shown in Figure 9.4, where the methods shown require a `ASTCDAssociation`, which is the AST node that represents an association.

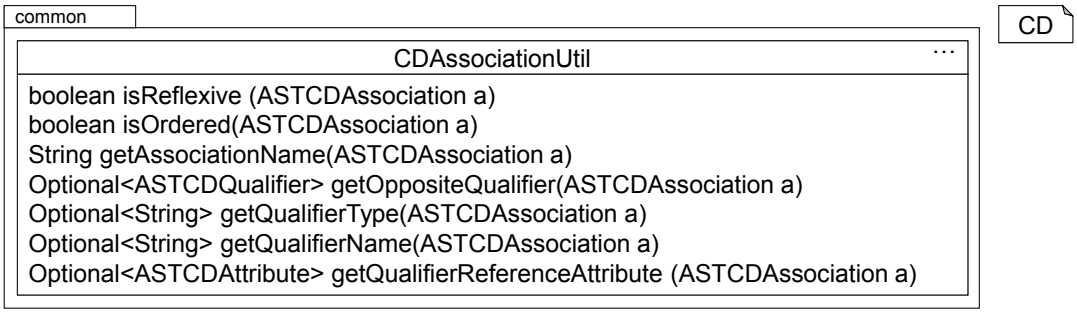


Figure 9.4: The methods provided by the `CDAssociationUtil` embedment helper to check for certain properties of associations.

- **`isReflexive(ASTCDAssociation s)`**: A reflexive association is an association that is bidirectional with the source and the target being the same. This methods checks if a given association fulfills this property.
- **`isOrdered(ASTCDAssociation s)`**: This method checks if the association is ordered in the given direction, i.e., the target of the association `s` has the «`ordered`»-stereotype.
- **`getAssociationName(ASTCDAssociation s)`**: The association name is derived from the associations properties as described in Section 4.2.4. The association's name is always lower case and if the association has cardinality `[*]` or `[1..*]`, the letter `s` is appended to the association's name.
- **`getOppositeQualifier(ASTCDAssociation s)`**: This methods checks whether the opposite of the given association is qualified. Technically, an association is considered to be qualified in one direction if `s` has a qualifier on the source of the association.
- **`getQualifierType(ASTCDAssociation s)`**: Returns the qualifier's type as a `String`. If the given association is not qualified, `Optional.empty()` is returned to denote a missing qualifier.
- **`getQualifierName(ASTCDAssociation s)`**: Returns the qualifier's name as a `String` only if the association is qualified. Otherwise `Optional.empty()` is returned showing a missing qualifier name.
- **`getQualifierReferenceAttribute(ASTCDAssociation s)`**: If a qualified association uses an attribute of the target as a qualifier, this method returns the referenced attribute. If no attribute is found or the qualified association uses a type qualifier, `Optional.empty()` is returned.

TransformationUtil Embedment Helper

The transformations used in the proposed approach are implemented using Java. Hence, support for handcoded extensions, design of run-time exceptions, or support for manually-written templates has to be provided. For such concerns, MontiDEx provides the `TransformationUtil` embedment helper, which is shown in Figure 9.5.

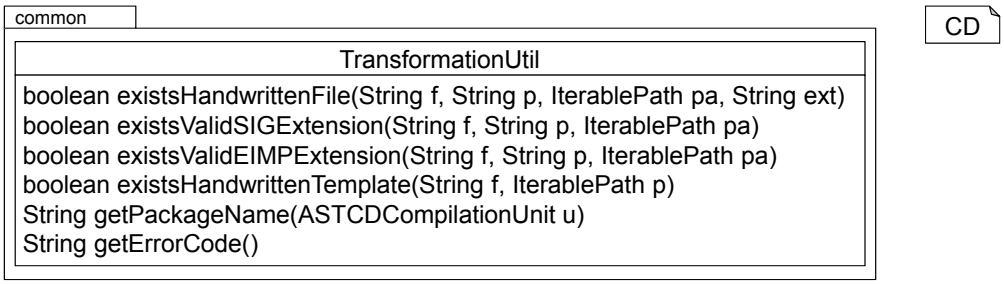


Figure 9.5: The methods provided by the `TransformationUtil` embedment helper to support handcoded extensions and creation of run-time exceptions.

- **`existsHandwrittenFile(String f, String p, IterablePath pa, String ext)`**: This method checks for a particular file during code generations. It is the essential method for realizing the handcoded extensions approach used by the MontiDEx code generator (cf. Chapter 6). Given a file name `f`, a package name `p`, a path `pa`, and a file extension `ext`, the method searches recursively in a path `pa` for a manually-written file named `f` having the extension `ext`.
- **`existsValidEIMPExtension(String f, String p, IterablePath pa)`**: This method is used to detect if a manually-written implementation extension for a particular class with the naming schema "`<Name>EIMP.java`" exist, where `<Name>` is an arbitrary name, exists. Given a file name `f`, a package name `p`, and a path `pa`.
- **`existsValidSIGExtension(String f, String p, IterablePath pa)`**: Similar to the previous method, this method is used to detect if an interface extension exists. Given a file name `f`, a package name `p`, and a path `pa`. The naming schema of interface extensions is "`<Name>SIG.java`", where `<Name>` is an arbitrary name.
- **`existsHandwrittenTemplate(String f, IterablePath pa)`**: The MontiDEx code generator supports manually-written templates to customize the generated source code. This requires the generator developer to define such possible extensions by using this method to detect manually-written templates. This method checks for a particular template with the name `f` in the path `pa`.

- **`getPackageName(ASTCDCompilationUnit u)`**: This method returns the package name for a given CD4Code-AST.
- **`getErrorCode()`**: The generated sources contain explicit spots in the source code, where exceptions have to be thrown, e.g., for derived attributes a `NotImplementedException` is thrown. Each thrown exception contains a unique error code, which is randomly generated each time the overall source code is produced. This helps tracing run-time exceptions. Hence, this method returns an unique error code as a `String` value.

ConstantsHelper Embedment Helper

Additional methods that may be required during code generation are the user's name, the generation time, and the generation prefix. These values are considered as constant during code generation and are provided by this embedment helper.

- **`getUserName()`**: Returns the name of the user generating the software system. Typically, this is the application developer or senior application developer (cf. Section 3.3).
- **`getGenerationTime()`**: The generation time is the time when the overall software system is generated. It is retrieved when an instance of the `Constants Helper` class is created. Hence, it is constant during the overall code generation process.
- **`getGenPrefix()`**: The generation prefix is the prefix used for methods that are additionally generated and required by the mapping in Chapter 5 to ensure association consistency but should not be used by the developers. The default prefix is "raw".

9.3 Methods for Code Generator Configuration

Even if a code generator is seen as a black-box by application developers, the predefined defaults and the code generator's behavior are not always suited. One simple example is changing the output folder, where the generated artifacts are placed. Another example is to generate only parts of the software system rather than the whole software system. Adaptation concerns of the generated artifacts also become necessary, whenever the intended changes to the generated source code affect multiple generated artifacts, e.g., implementing a certain interface "to streamline the generated code for specific target environments" [Sel03]. In this case, the proposed handcoded integration approach (cf. Chapter 6) is not practical.

In this thesis, configurability of a code generator is regarded as the choice a (senior) application developer has to influence the code generator's behavior. Such configuration options are predefined and designed by a generator developer and offer a limited degree of flexibility, because they are typically either "switches" that can be turned on or key-value pairs that can be set. Hence, the MC framework offers a script-based approach to configure the code generator in a more detailed way (cf. Section 2.2). In particular, it uses a Groovy Script to define language processing, context condition checking, and code generation. When combining this script-based approach with the synergetic transformation- and template-based code generation (cf. Chapter 8), a code generator allows application developers to arbitrarily select transformations that are executed by the code generator to define the functionality of the generated software system. For example, the MontiDEx architecture consists of MontiDEx modules (cf. Section 9.2), that can be selected in a Groovy script to generate, e.g., only the application layer.

Configuration of a code generator allows for high-level decisions but is not sufficient to change a code generator's internals. Hence, adaptability addresses the adaptation of code generation internals without changing the code generator's source code. Note that code generator extensions, i.e., manually-written transformations and templates that are added to the code generator without adapting existing ones, can be executed directly or even implemented in the configuration script.

9.3.1 Technical Realization of MontiDEx Configurations

The technical realization of MontiDEx configurations are located in the `configuration` package (cf. Section 9.2). It consists of Java classes with predefined configuration options and methods that can be used in a Groovy script. By default, a script to generate all layers of a data-centric application and the following predefined scripts are provided: developer mode with additional reporting (`dev`); Plain-Old-Java-Objects (POJOs) (`pojo`); and generate implementation concerns from ADJava models (`ad`), which are further explained in Chapter 10. An overview of the package is shown in Figure 9.6.

The `DexConfiguration` class provides all configuration options of the code generator that are used by the code generator at generation time. An instance of this class is created before the code generator is executed based on the parameters passed to the code generator. The following configuration options are supported. Note that for presentational reasons only the accessors are explained.

- **`getModel()`**: Returns a reference to the input model passed to the code generator.
- **`getModelPath()`**: Returns a list of paths, where models are located.
- **`getOut()`**: Returns the output directory.
- **`getHandcodedPath()`**: Returns the list of paths that may contain manually-written extensions for the code generator.

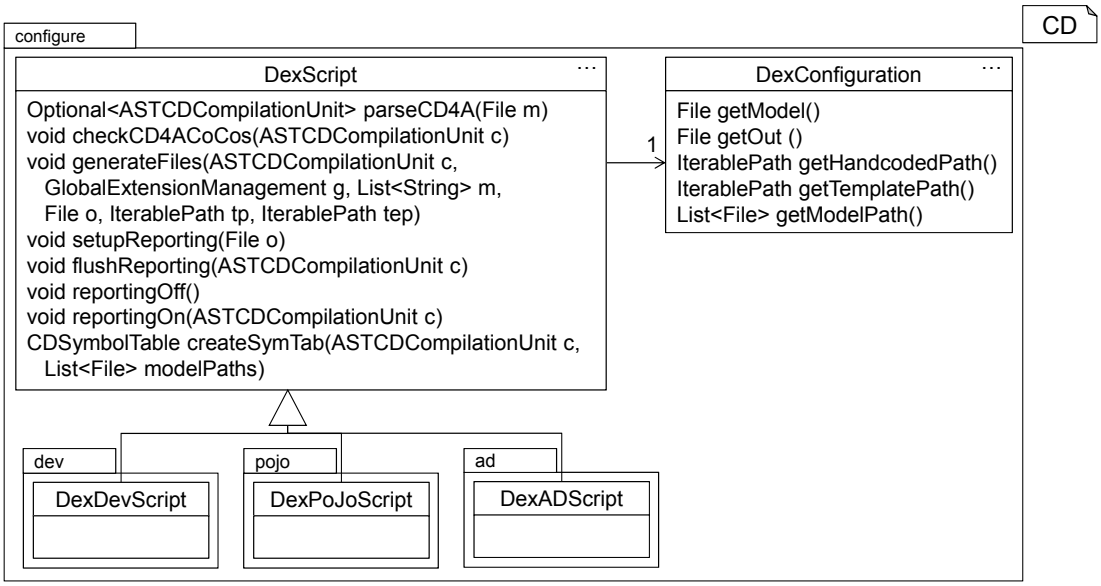


Figure 9.6: The Java classes provided by the `configure` package and its subpackages.

- **`getTemplatePath()`**: Retrieve the list of paths that may contain manually-written template extensions for the code generator.

The `DexScript` class is used by each Groovy script to access predefined functionality to perform language processing, check context conditions, and generate source code. In addition, it is also used to define default behavior of the code generator. Each default is specified in a subclass executing predefined Groovy scripts. For example, the developer configuration script is located in the `dev` package (cf. Figure 9.6) and contains the predefined `DexDevScript` subclass that provides additional functionality and default values. Another example is the `DexPoJoScript` default behavior specified for Java POJOs. Subsequently, all methods provided by the `DexScript` class are explained.

- **`parseCD4A(File m)`**: Parse the CD4A model in artifact `m` and return the root AST node. If the model cannot be processed, `Optional.empty()` is returned.
- **`checkCD4ACoCos(ASTCDCompilationUnit c)`**: This method is used to check context conditions. By default, it checks all registered context conditions but allows to add manually-written context conditions that are checked as well.
- **`generateFiles(ASTCDCompilationUnit c, GlobalExtensionManagement g, List<String> m, File o, IterablePath tp, IterablePath tep)`**: This method executes the code generation for a particular CD4Code-AST `c`. It requires a path to the model `m`, which may contain multiple CD4A

models; an instance of the `GlobalExtensionManagement`, which is responsible to handle global values and adaptations of the code generator process at run-tume as explained in Section 9.3; an output directory `o`; a path containing manually-written extensions `tp`; and a path that contains manually-written templates `tep`.

- **`setupReporting(File o)`**: The MontiDEx code generator provides a set of reporting options, as described in Section 9.4. This method allows to setup reporting by specifying the output directory `o` to which all files are written.
- **`flushReporting(ASTCDCompilationUnit c)`**: Reporting can be flushed if it is not used anymore. This means that all reporting for a particular parsed model `c` is removed.
- **`reportingOff()`**: Disable reporting temporarily.
- **`reportingOn(ASTCDCompilationUnit c)`**: Enable reporting if it has been setup correctly by calling the `setupReporting()`-method.
- **`createSymTab(ASTCDCompilationUnit compUnit, List<File> modelPaths)`**: Create a symbol table for a parsed CD4A model. Model paths have to be added if the parsed CD4A model references other CD4A models.

An example of a Groovy script using the predefined configuration options and functionality is shown in Listing 9.1. Typically, a configuration script starts by parsing an input model and checking context conditions, as shown in ll.2-9. Afterwards, transformations defined in MontiDEx modules are executed to manipulate the CD4Code-AST (ll.12-13) and, finally, the files are generated (l.16).

Groovy

```
1 // parse input model
2 cdAst = parseCD4A(model)
3 if (!cdAst.isPresent()) {
4     error("Failed to parse " + model)
5     return
6 }
7
8 // check context conditions
9 checkCD4ACoCos(cdAst.get());
10
11 // execute a transformation
12 ApplicationCore a = new ApplicationCore();
13 a.transform(cdAst.get())
14
15 // generate files
16 generateFiles(cdAst.get(), modelPath, out)
```

Listing 9.1: A configuration script to parse a model (ll.2-6), check context conditions (l.9), execute a transformation (ll.12-13), and generate files (l.16).

By default a set of global variables are predefined to access the arguments passed to the code generator. An overview of all predefined variables is given in Table 9.2.

Predefined Variable Name	Description
`model`	The reference to the input model.
`modelpath`	A list of paths where models are located.
`out`	The path to the output directory, where the generated files should be placed.
`handcodedPath`	A list of paths, where manually-written source code is located.
`templatePath`	A list of paths, where manually-written templates are located.

Table 9.2: An overview of the set of variables that are predefined to be used in a MontiDEx configuration script.

In general, an advantage of using a scripting language to configure the code generator is that the execution flow of the code generator can be defined and errors can be customized. For instance, as shown in ll.3-6 in Listing 9.1 the script allows to handle errors that may occur during parsing.

9.4 MontiDEx Reporting Facility

In general, "model-error reporting and debugging facilities must accompany practical automatic code generators" [Sel03]. Such debugging facilities help in detecting modeling errors and code generator errors. In MontiDEx debugging is supported by reports, each of which presents structured and aggregated information about the code generation process. Such traces and debugging information has already proven to be beneficial for code generators [Jör13].

In the remainder of this section, all reports supported by MontiDEx are introduces and explained in detail.

9.4.1 Textual Reports

We distinguish between *log reports*, which are reports comparable to log files containing information about a certain part of the code generation process, and *aggregation reports*, which aggregate information regarding the code generation process. Each report provided by MontiDEx is located in the `reports` folder and explained in Table 9.3.

Report	Description
01_Summary.txt	The overall summary presents an aggregated report on the entire code generation process. It contains the number of errors, warnings, generated files, instantiations of embedment helpers, and templates includings.
02_GeneratedFiles.txt	This aggregated report lists all generated source code files. It, additionally, links the template used for generating a particular source code artifact and the corresponding CD4Code-AST node.
03_HandwrittenCodeFiles.txt	Because the code generator supports handcoded extensions, this report summarizes all found manually-written source code files. If the code generator detects a manually-written extension but cannot use it for the generated source code (because it is not a subclass of a generated file or a super interface), then these files are listed as well.
04_Templates.txt	During the code generation process, not all templates of the code generator are used, because the model may not contain certain concepts. This log report lists all templates used for code generation for a particular input model and how often they have been called. Nevertheless, this report lists all unused templates as well. Furthermore, it lists all manually-written templates located in the template path (cf. Section 9.3.1).
05_HookPoint.txt	This log report lists all hook points and the operations that have been executed, i.e., the hook point was replaced, added before, added after, or simply called with out any effect. Moreover, the type of the value of each hook point is reported.
06_Instantiations.txt	This log report contains all embedment helpers that have been instantiated during the code generation process as well as the number of instantiations.

07_Variables.txt	During the code generation process, global variables can be used to exchange information between templates (cf. Section 2.2.4). These variables are listed in this log report. In addition, it contains the number of times the value of the global variable has been changed.
08_TemplateTree.txt	This report shows the executed templates in a tree-like structure. Additionally, it contains all variable assignments, embedment helper instantiations and hook point executions.
09_NodeTree.txt	The node tree log report represent the CD4Code-AST used for code generation. It contains information about how often a particular CD4Code-AST node has been used by templates for generating source code.
10_TypesOfNodes.txt	A summary of the CD4Code-AST used for code generation focusing on CD4Code-AST node types is listed in this log report. It contains the number of AST objects of a certain AST type and how often a particular CD4Code-AST type has been used by templates for code generation.
11_SymbolTable.txt	Besides the CD4Code-AST, the symbol table is used during code generation to store and retrieve information. This report lists the content of the symbol table after the CD4Code-AST has been used for code generation.
12_Transformations.txt	This report visualizes the effects - modifications of the CD4Code-AST and attachments of templates - as a list in textual form.
13_Detailed.txt	This report is a summary of the overall code generation process comprising all other reports. It provides a detailed log of all occurred events.

Table 9.3: A list of all log and aggregation reports provided by MontiDEx.

9.4.2 Graphical Report

Besides textual reports, graphical reports of the code generation process are generated showing interconnections between templates and embedment helpers. Templates and

embedment helpers are grouped w.r.t to their package structure and the number of embedment helper instantiations and template calls is shown.

An example is depicted in Figure 9.7. It shows templates (indicated by purple or green circles), the template's name, and number of template calls. For instance, the `Class.ftl` template is called 51 times. The result of the templates marked green is written into a file (`Class.ftl` and `Interface.ftl`), whereas the results of the purple-colored templates (`Constructor.ftl`, `Attribute.ftl`, `Generics.ftl`, `Method.ftl`, and `InterfaceMethod.ftl`) are returned to the calling template. An arrow between two templates denotes a direct template call and the number next to an arrow denotes the number of calls, e.g., the template `Constructor.ftl` has been called 51 times by the `Class.ftl` template. Depending on the number of calls, the thickness of the arrow changes. The more often a template is called, the thicker an arrow is displayed.

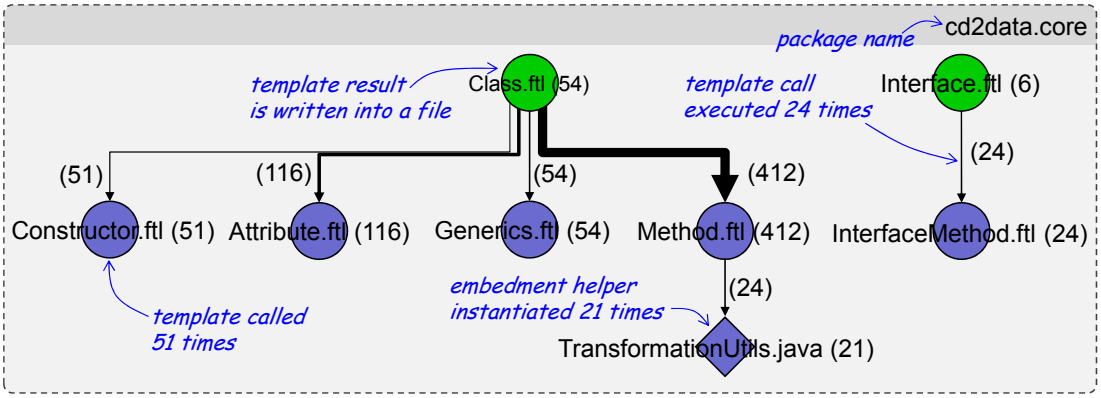


Figure 9.7: A graphical representation of templates and embedment helper interrelation showing the amount of times each of which is called.

Embedment helpers are depicted using a purple diamond symbol, e.g., `TransformationUtils.java` in Figure 9.7. The number next to the embedment helper's name indicates the total number of embedment helper instantiations. An arrow from a template to an embedment helper shows the number of calls. For instance, the `TransformationUtils.java` embedment helper has been instantiated 21 times and called 24 times.

A dashed bounding box around templates and embedment helpers denotes that they are grouped into one MontiDEx module, which is represented by a Java package. The name of the package is shown in the top right corner of the bounding box.

9.5 Method for Adapting and Deploying MontiDEx

To reduce the conceptual break introduced by an MDD approach, the tools have to be seamlessly integrated into existing software development environments (cf. [Sel03]).

Therefore, a usage methodology of the MontiDEx code generator has been proposed in Section 3.3. However, it still leaves open the question, when to use what approach to adapt the code generator. Hence, in the remainder of this section, we address this question. Moreover, we elaborate on how to deploy the different generated products.

9.5.1 Method for Adapting the MontiDEx Code Generator

The MontiDEx code generator provides a variety of customization and extension mechanisms as introduced in Section 8.2.4 and in Section 9.3, each of which is particularly designed for one type of customization or adaptation. To guide generator developers and senior application developers in choosing the right mechanism, a method is provided. Note that the method presented is designed for extensions of existing functionality of the code generator. In particular, if, e.g., additional classes, variables, or interfaces have to be created in the generated source code or well-formedness checks are required by the code generator and the added functionality can be generalized in terms of object-oriented concepts, then transformations have to be created as described in Section 8.4.1. Moreover, manually-written extensions of the generated source code are not considered, because a method of their use is explained in Section 6.4.

The proposed methodology is shown in Figure 9.8. The first step is to identify if the extension affects only one product artifact or multiple. If the adaptation is local, i.e., affects only one file, then template attachments can be used to adapt CD4Code-AST-specific templates. If customizations are global, i.e., affect multiple artifacts, then, at first, the set of hook points has to be checked. If no hook point exist, then template extensions are used to either add other templates or replace a template.

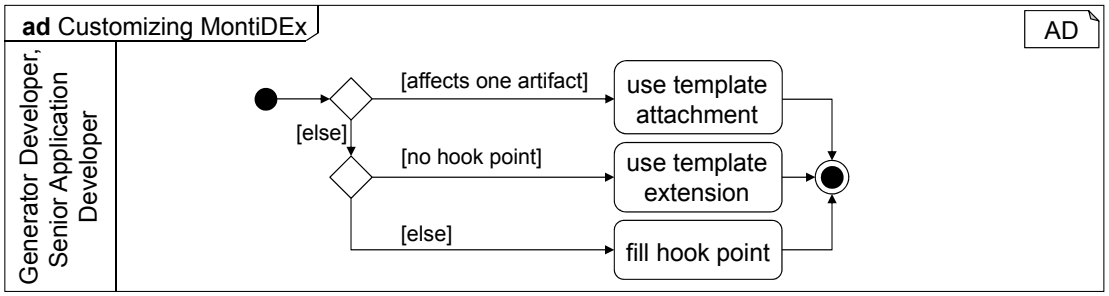


Figure 9.8: A method for the adaptation approaches of the MontiDEx code generator.

9.5.2 MontiDEx Project Types and Deployment

Due to the configuration and adaptation mechanisms provided, MontiDEx allows to synthesize a variety of products. Such MontiDEx products can be deployed as different

projects according to their intended use and users. We distinguish between the following different projects:

Developer Project: This version of the project contains the CD4A model, the generated Java source code, and the handcoded extensions as well as the source code for the persistence infrastructure. It is primarily intended for application developers, who plan to continue development of the MontiDEx product. Note that the source code of the MontiDEx code generator is not part of the MontiDEx product.

Client Model Development Project: This version contains only the CD4A model, the generated Java source code and the handcoded extensions. The persistence infrastructure is not necessary, as only the application is further developed.

Client Java Development Project: This version consists of the generated Java source code only but is not included the CD4A model. The goal of this project is to continue development of the application without adapting the CD4A model.

Client Project: This version consists of only the executable MontiDEx product that is started. It is primarily intended for end users.

Regardless of the project, a technical infrastructure is required in order to execute the data-centric application or continue development. In particular, each MontiDEx product requires Java version 1.8 and a persistence infrastructure supporting the functionality described in Section 7.5.

After setting up a technical infrastructure, the MontiDEx product is deployed using the following steps:

1. **Infrastructure Setup**.
   Set up the application server in the targeted usage environment of the end user or application developer. In addition, set up a the database server. Technically, it can be either the same server or a different one. If an existing infrastructure is reused, this step can be skipped.

2. **Product Generation and Packaging**.
   Build the chosen version of the MontiDEx product. Before packaging the product, the generated source code has to be manually adapted to fit into the created infrastructure, i.e., the corresponding IP addresses have to be set. Afterwards, the source code is packaged into an executable Java Archive (JAR) file and deployed to the application server. Packaging is done via the Maven [www16c] build automation tool. Deployment on the application server is necessary whenever the source code of the generated product changes.

   If a version of the MontiDEx product has previously been deployed, data migration can be used to migrate already used databases Section 7.6.

Chapter 10

Case Example: Extended Infrastructure for Process Automation

The data-centric infrastructure, which is described in Chapter 7, supports management of data structures. Application-specific behavior enabling process automation [AHW03], which is regarded as automated execution of CRUD operations on a data structure in this thesis, has to be added via manually-written customizations of the generated source code (cf. Chapter 6) or adaptations of the code generator (cf. Section 9.3).

In this thesis, an *extended data-centric infrastructure* for process modeling and execution aims to reduce manual implementation effort that is required to realize process automation. It enables adaptation of processes after deployment of a data-centric application, where a code generator is not available. This is achieved by explicit processes modeling using the ADJava ML. This ML has been developed in this thesis and is based on UML ADs with a simplified control flow notation and auto-connect capabilities (cf. *MR-2*). Each ADJava model is executed by an interpreter, which is part of the extended data-centric infrastructure, to facilitate rapid adaptation of processes for data-centric applications by a modeler, who is unaware of implementation concerns (action implementations, guards and conditions). Implementation concerns are added by an application developer via manually-written Java source code extensions of the interpreter. Alternatively, ADJava models can be enriched with Java source code to realize implementation concerns. If Java source code is used in an ADJava model, code generation is used to generate Java source code that is executed by the interpreter at run-time.

The goal of this chapter is to present a method to implement an extended infrastructure for process modeling and execution to evaluate the proposed customization and adaptation mechanisms of MontiDEx. The extended infrastructure extends the data-centric infrastructure. Hence, this chapter is structured as follows. First, general considerations and requirements for the extended infrastructure are presented in Section 10.1. Afterwards, the ADJava ML is introduced in Section 10.2. Next, the integrated approach to execute ADJava models is presented in Section 10.3, which is based on previous work [LN16]. Afterwards, a method for development of processes with ADJava models is provided in Section 10.4. Finally, the customization and adaptation mechanisms used for the realization are evaluated and limitations are discussed in Section 10.5.

10.1 General Considerations and Requirements

Besides the general requirements presented in Section 3.2, the following particular requirements and considerations for an extended data-centric infrastructure for process automation can be identified. They are derived from the described methods for MDP and MDD of data-centric applications (cf. Section 3.3).

MR-2-1 Control and object flow chaining: To simplify the notation of control and object flows, ADJava supports sequences of control and object flows.

MR-2-2 Pin and type auto-connect: Each input and output pin has to be connected manually to denote object flows. To simplify this notation pin- and type-based auto-connect capabilities are provided for input and output pins.

MR-2-3 Explicit roles modeling: To structure the overall process into partitions of actions that have to be executed by certain end users, ADJava supports role partitions.

GR-4-1 Stepwise execution: To facilitate the envisioned MDP of data-centric applications (cf. Section 3.3.1), execution of an ADJava model can be interrupted at any point in time.

GR-4-2 Interpretation of ADJava models: To support rapid prototyping by modelers, ADJava models are interpreted to facilitate rapid changes performed by modelers.

GR-4-3 Integration of handwritten implementations: Implementation concerns are added by the application developer via Java source code that extends the interpreter and is executed at interpreter run-time. This approach for handling implementation concerns is provided to support deployment of MontiDEx products without the MontiDEx code generator (cf. Section 9.5.2).

GR-4-4 Code generation from ADJava models: Implementations of actions, guards, and conditions are also supported on the model-level via language composition [Sar06, Rei15]. Therefore, the ADJava ML is a the composition of a base AD ML, which is introduced in the remainder of this chapter, and the JavaDSL, which is a DSL for the Java language. To execute Java source code embedded in ADJava models, an extension for the MontiDEx code generator is provided. This code generator extension transforms the JavaDSL part of a ADJava model into executable Java source code, which is executed at interpreter run-time. This approach aims to reduce the implementation effort required by application developers to extend the interpreter to implement actions, guards, and conditions (cf. *GR-4-3*).

GR-5-1 GUI for managing ADJava model execution: The execution of ADJava models is controlled via a GUI. It is an extension of the GUI provided by the data-centric infrastructure (cf. Section 7.3).

GR-5-2 Execution state (de)serialization: When the execution of an ADJava model is interrupted, the current execution state can be serialized. Likewise, serialized states can be deserialized to continue execution.

10.2 ADJava: Activity Diagram Modeling Language

Before presenting the ADJava ML, we briefly introduce the main concepts of UML AD because ADJava is based on UML AD. Interested readers are advised to refer to [Boc03a, Boc03b, Boc03c, Boc04a, Boc04b, Boc05] for a detailed introduction.

The example in Figure 10.1 shows a transaction submission process. Each `Customer` creates a transaction. This transaction is validated in the `validate credit` action to ensure that the customer is able to pay for the transaction. If the customer is unable to pay, he is notified and the process ends. Otherwise, if the customer is able to pay, the `Accountant` validates (`validate transaction` action) if the receiver of the transaction is valid. If the receiver is valid, the transaction is executed and a `fraud check` action is started. Alternatively, if the receiver is not valid, the customer is notified to enter a correct receiver. After the transaction has been executed and the fraud check has been performed, it is checked whether the transaction has been successfully executed and the fraud check has not been positive. If the result is valid, then the transaction is finalized and the process is terminated. Otherwise, if the result of processing a transaction or fraud checking returns an error or violation, the customer is notified and the transaction is not finalized and the customer is notified.

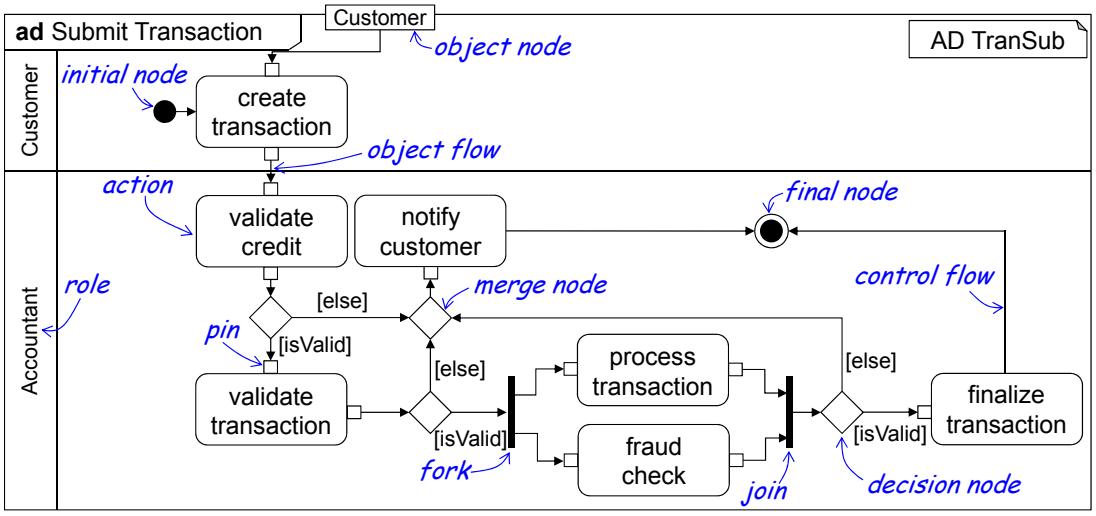


Figure 10.1: A UML AD defining the actions for submitting a transaction.

Figure 10.1 also shows the main elements of a UML AD model. Each activity starts in the *initial node* (black dot) and ends in a *final node* (circled black dot). In between, *actions* are executed, e.g., "notify customer", that are connected by a *control flow* or an *object flow*. Besides actions, *decision nodes* can be used to indicate conditions and merged by using *merge nodes*. For parallel execution of actions, *fork nodes* can be used, which have to be joined by *join nodes*. Moreover, *object nodes* are abstract typed actions denoting an object flow. Inputs and outputs of actions they are called *pins*.

Technically, the ADJava ML is realized with the MC framework (cf. Section 2.2). The full description of the language including the grammar and context conditions is shown in Section C.2 and Section E.3. As shown in Figure 10.2, the base activity language (AD), which realizes the main concepts described in the example in Figure 10.1, is extended with a the JavaDSL, which is a DSL for Java, to define guard, condition, and action implementations.

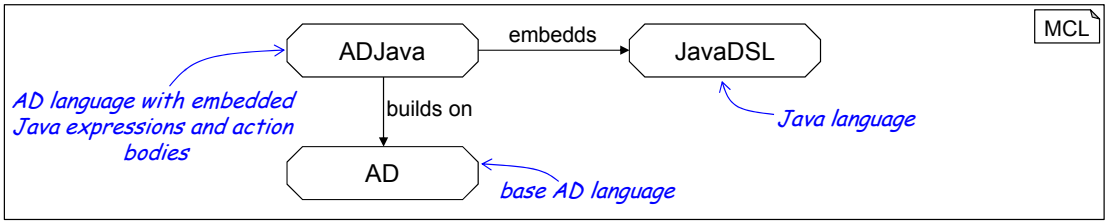


Figure 10.2: Overview of the developed AD languages.

In the remainder of this section, the example in Figure 10.1 is used to present the ADJava ML. The full textual model of the example is shown in Listing D.5.

10.2.1 Activity Definition

An ADJava model is defined in one single file that has the contained activity diagram's name. This design decision enables traceability during language processing (cf. [Sch12]). Each ADJava model can contain a package declaration for structural purposes. A package declaration is defined with the **`package`** keyword and a full qualified name. In addition, an import statement can be defined to reference external data types. An import statement is defined with the **`import`** keyword and the qualified name of the external data type. In general, each external data type used in an ADJava model has to be imported to ensure referential integrity (cf. [Sch12]).

The main element of an ADJava model is the activity definition, which defines exactly one activity. An activity definition starts with the **`activity`** keyword followed by a name. Each activity can have arbitrary many input and output pins and encloses actions, object nodes, and control and object flow definitions specified in arbitrary order.

For example, Listing 10.1 shows the `SubmitTransaction` activity (l.3) with the `dex.activities` package declaration (l.1) and the `dex.activities.Customer`

imported external data type (l.2). The `SubmitTransaction` activity has one input pin (l.3), which is defined by a type (`Customer`) and a name (`c`) between two brackets.

```
1 package dex.activities;                                    ADJava TranSub
2 import dex.activities.Customer;                                  ...
3 activity SubmitTransaction(Customer c){
4 }
```

Listing 10.1: An example of a simplified ADJava model.

Besides input pins, each activity can have arbitrary many output pins, each of which is specified after a colon (**:**)-symbol as a comma separated list with a type and name for each output pin. For instance, Listing 10.2 shows multiple input pins separated by a comma (`Order o, Customer c` in l.1) and multiple output pins (`Notification n, Task t` in l.2).

```
1 activity MultipleParams (Order o, Customer c)              ADJava
2                         : Notification n, Task t {
3 }
```

Listing 10.2: An ADJava model with multiple input and output pins.

An activity can also have a pre- and a postcondition. A precondition defines a condition that has to be fulfilled before the activity is executed. A postcondition is a condition that has to be fulfilled after an activity is executed. Preconditions are boolean expressions between square brackets (`[...]`). Likewise, postconditions are boolean expressions between double square brackets (`[[...]]`). Technically, in the ADJava ML the boolean expressions are Java expressions. However, it is possible to embed arbitrary expression languages in the AD base language (cf. Section 10.2).

An example of an activity with a precondition and a postcondition is shown in Listing 10.3. The precondition is defined in l.1 (`[ c != null]`). The postcondition is in l.5 (`[[ t != null]]`).

```
1 [ c != null ]                                              ADJava TranSub
2 activity SubmitTransaction(Customer c)                           ...
3                           : Transaction t {
4 }
5 [[ t != null ]]
```

Listing 10.3: Each activity can have a precondition between `[...]` (l.1) and a postcondition between `[[...]]` (l.4).

10.2.2 Actions

Actions define the behavioral units of an activity. Each action is defined by the **action** keyword and a name. Just as an activity definition (cf. Section 10.2.1), an action can have arbitrary many input and output pins as well as one precondition and one postcondition. Moreover, an action's body can be a Java implementation.

For example, Listing 10.4 shows the `CreateTransaction` action (ll.1-2), which has one input pin (`Customer c`) and one output pin (`Transaction t`). Furthermore, this example shows the `ValidateCredit` action (ll.5-6), which has an input pin (`Transaction t`) and an output pin (`Transaction o`) as well as a precondition (`[t != null]` in l.4) and a postcondition (`[[ o != null ]]` in l.5). The last action in this example is the `NotifyCustomer` action, which shows an action body implementation using Java (l.8).

```
1 action CreateTransaction(Customer c)                    ADJava TranSub
2                           : Transaction t;                    ...
3
4 [t != null]
5 action ValidateCredit(Transaction t)
6                       : Transaction o;
7 [[ o != null ]]
8
9 action NotifyCustomer(Transaction t) {
10   NotificationService.notify(t.getFrom().getCustomer()).send();
11 }
```

Listing 10.4: Actions are defined with the **action** keyword and a name. They can also have input and output pins (ll.1-2 and ll.5-6), pre- and postconditions (l.4 and l.7), and Java implementations (l.10).

To reuse existing activities, actions may call other activities. This is denoted by the **execute** keyword in an action's body followed by a qualified activity name. If the called activity has input and output pins, each input pin of the action has to be mapped to an input pin of the called activity. Likewise, each output pin of an action has to be mapped to an output pin of the called activity.

For example, Listing 10.5 shows two actions, each of which calls an activity. The `FinalizeTransaction` action (ll.1-3) calls the `coreactivities.ExecuteTransaction` activity. The action's input pin (`Transaction t` in l.1) is mapped to the called activity's input pin using the action pin's name (l.2). The second action (`FraudCheck` in ll.5-7) calls the `security.CheckValidity` activity (l.6). Again the input pins are passed to the called activity (l.6) but, in addition, the action's output pin

(`Transaction o`) is mapped to the activity's output pin (`r`). This is done by addressing activity's output pins via a dot (.) and mapping them via the arrow (->)-symbol, as shown in l.6. In this example, the output pin `r` is mapped to the output pin `o`. Multiple output pins are addressed the same way but using a comma separated list. Hence, multiple output pins are mapped element by element in the order they are defined.

```
1 action FinalizeTransaction(Transaction t){                    ADJava TranSub
2   execute coreactivities.ExecuteTransaction(t);                    ...
3 }
4
5 action FraudCheck(Transaction t): Transaction o{
6   execute security.CheckValidity(t).r -> o;
7 }
```

Listing 10.5: Example of two actions calling activities.

10.2.3 Object Nodes

Object nodes are typed and abstract activity nodes that are used to define object flow between actions. For example, the object flows in the examples in Section 10.2.1 are defined via input and output pins. In addition, it is possible to define object nodes that are shared between actions. Such object nodes are defined with a preceding **`data`** keyword followed by a type and a name. Each shared object node can be accessed from any action contained in the shared object node's enclosing activity. Conceptually, shared object nodes are based on central buffers and data stores, which are defined in the UML AD specification [www15b].

For example, Listing 10.6 shows the shared object node `t` of type `Transaction`.

```
1 activity ADObjectNodes{                                       ADJava
2   data Transaction t;
3 }
```

Listing 10.6: An example of a shared object node that can be used by actions within the enclosing activity.

In general, shared object nodes are not considered as input and output pins of the enclosing activity. Hence, when an activity with enclosed shared object nodes is called from an action (cf. Section 10.2.2), the action's input pins cannot be mapped to the called activity's shared object nodes.

10.2.4 Control And Object Flow

A control flow in an activity diagram describes a flow of control in the stepwise execution of actions. It connects two actions or *control nodes*, which coordinate the flow between other nodes, with a directed edge denoted by the (**->**)-symbol. For instance, Listing 10.7 shows a control flow between `Action1` and `Action2` in l.1. To simplify the notation of control flows, multiple control flows can be chained (cf. *MR-2-1*), as already proposed [www16j]. Note that this represents an extension to the UML AD ML designed in [Rei15], where control flows are defined between two nodes only, as shown in l.1 in Listing 10.7.

```
1 Action1 -> Action2;                                    ADJava
2 Action3 -> Action4 -> Action5;
```

Listing 10.7: An example of control flows, which define flows of control between action and control nodes.

Besides control flows, object flows denote flows can be used to of objects between two nodes (actions and control nodes). An object flow represents a special type of control flow and, hence, is defined similarly, i.e., directed edge using the (**->**)-symbol. In addition, each object flow specifies the mapping of an output pin to an input pin. An output pin is mapped to multiple input pins, when a decision or fork node is used.

For instance, Listing 10.8 shows the definition of an object flow (l.4) from `Action1` to `Action2`. It maps the output pin `t` of `Action1` to the input pin of `Action2` with the same name.

```
1 action Action1 : Type t;                               ADJava
2 action Action2(Type t);
3
4 Action1.t -> Action2.t;
```

Listing 10.8: An example of an object flow definition (l.4).

The flow of control can explicitly be structured by different control nodes types. In the remainder of this section, the different control node types supported by the ADJava ML are explained.

Control Node Types

For presentational reasons, the presented examples of control node types show control flows only. However, control nodes can also be use for object flows.

Initial Node An initial node marks the start of the activity's execution flow. It is defined by the keyword **`initial`** keyword, as shown in Listing 10.9. Initial nodes have no ingoing edges and only one outgoing. It is possible to have multiple initial nodes as well as none.

```
1 initial -> CreateTransaction;                                    ADJava
```

Listing 10.9: Initial nodes are denoted by the **`initial`** keyword and have to be connected to one other node.

Final Node Final nodes denote the end of a flow in an activity. Hence, they have one ingoing edge and no outgoing edges. Each final node is defined by the **`final`** keyword, as shown in l.1 in Listing 10.10.

```
1 Action1 -> final;                                                ADJava
2 Action2 -> flowfinal;
```

Listing 10.10: Final nodes are denoted by the **`final`** keyword and do not have any outgoing edges.

In addition, to denote the end of a flow only rather than the end of a flow in an activity, flow final nodes are supported [www15b]. Each flow final node is defined by the **`flowfinal`** keyword as shown in l.2 in Listing 10.10.

Fork Node A fork node marks parallel execution of actions. It has one incoming edge, which can be a control flow or an object flow. Each control flow can be defined in a control flow or an object flow by using the **`||`**-symbol, e.g., l.1 in Listing 10.11. Alternatively, an explicit fork node, which is defined by the **`fork`** keyword (l.3) and a name, can be used. Such explicit fork nodes can be used in other control or object flows as well (ll.4-5).

```
1 Action1 -> (Action2 || Action3);                                 ADJava
2
3 fork f1;
4 Action4 -> f1;
5 f1 -> (Action5 || Action6);
```

Listing 10.11: Fork nodes are either define implicit by the **`||`**-symbol (l.1) or explicit by the **`fork`** keyword and a name (l.3).

Join Node Join nodes allow to synchronize control and object flows previously forked by fork nodes. A join node has multiple incoming edges but only one outgoing edge

and is defined similar to a fork node, i.e., implicit in a control flow or an object flow using the **||**-symbol or explicit as a join node defined by the **join** keyword and a name.

For example, l.1 in Listing 10.12 shows an implicit definition, whereas ll.3-5 show an explicit definition and its use. In contrast to fork nodes, the implicit definition uses the **||**-symbol on the left-hand side of a control flow.

```
1 (Action1 || Action2) -> Action3;
2
3 join j1;
4 Action4 -> j1;
5 Action5 -> j1;
```

ADJava

Listing 10.12: Join nodes are defined implicitly on the left-hand side of a control or object flow or explicit by the **join** keyword and a name.

Decision Node A decision node denotes a branch in a control flow or an object flow that depends on predefined guards. Each decision node can have one or two incoming edges and has two outgoing edges. For each outgoing node a guard has to be defined. If a guard is true, the flow continues at this edge. Otherwise, the other guard is evaluated. Syntactically, a decision node is defined either implicit using the **|**-symbol or explicit using the **decision** keyword and a name.

For example, Listing 10.13 shows the implicit definition of a decision node in l.1. The guard for `Action2` is `[t != null]`. The guard for `final` is `[else]`, which is a predefined guard that can only be used if another guard that is not an `[else]` guard is defined. Moreover, in l.3 an explicit definition is shown, which is used within other control or object flows as shown in ll.5-6.

```
1 Action1 -> [t != null] Action2 | [else] final;
2
3 decision d;
4 Action2 -> d;
5 d -> [t!= null] Action1;
6 d -> [else] final;
```

ADJava

Listing 10.13: Decision nodes are implicitly defined by a **|**-symbol and guards (l.1) or explicitly by the **decision** keyword and a name (l.3).

Merge Node Similar to join nodes, merge nodes join control and object flows to one single outgoing control or object flow. However, because there is no synchronization

between the incoming flows, merge nodes should not be used to join concurrent control and object flows created by fork nodes.

Following the general idea of implicit and explicit notion, a merge node can be defined similarly. For an implicit definition the `|`-symbol is used on the left-hand side of the flow definition as shown in Listing 10.14 in l.1. For an explicit definition, the **`merge`** keyword followed by a name can be used (ll.3-5.).

```
1 (Action1 | Action2) -> Action3;                    ADJava
2
3 merge m:
4 Action1 -> m;
5 Action2 -> m;
```

Listing 10.14: Merge nodes are defined implicit by the `|`-symbol on the left hand side (l.1) or explicit by the **`merge`** keyword and a name (l.3).

10.2.5 Roles

In general, activity diagrams support partitions, which groups actions with common characteristics [www15b]. Semantically, they provide certain constraints on the invoked action in the activity diagram. The ADJava ML developed in this thesis supports role partition only (cf. *GR-2-3*). Such a partition groups actions that are assigned to a particular role. Only this role is allowed to execute the actions in this partition.

Each role partition is defined with the **`role`** keyword followed by a name and a comma separated list of actions that belong to this partition enclosed between `{...}`. For example, Listing 10.15 defines the role `user` in ll.1-3, which consists of only the `Action1, Action2` actions (l.2). Hence, only the end user having the role `user` can execute the actions `Action1` and `Action2`.

```
1 role user{                                         ADJava
2   Action1, Action2;
3 }
```

Listing 10.15: Role partitions are defined by the **`role`** keyword, a name (l.1), and an action sequence (l.2).

10.2.6 Pin and Type Auto-Connect

To simplify ADJava model definitions by neglecting explicit mapping of input and output pins, and object nodes, auto-connect capabilities based on MontiArc [Hab15] are

provided (cf. *GR-2-2*). In particular, ADJava ML supports *pin connect* and *type connect*. Pin connect is designed to connect two non-referenced object nodes with the same type and name. Type connect connects input and output pins with the same type. Both auto-connect modes are defined by the **autoconnect** keyword followed by either the **pin** or the **type** keyword for the modes with the same name. However, only one auto-connect mode can be chosen in one ADJava model.

These modes are realized by connectors [LN16], which define how a mapping of input and output pins is performed. Connectors can be arbitrarily selected and chained. In the remainder of this section, all provided connectors are explained.

Call Behavior Connector

The call behavior connector simplifies the definition of call behavior actions (cf. Section 10.2.2) by connecting input and the output pins. For the type auto-connect mode, the input pins and the output pins with the same type are connected. However, this is not always possible if the same type is used multiple times. For the pin auto-connect mode, the input pins and output pins with the same name and type are connected.

For example, Figure 10.3 (at the top) shows an explicit mapping of input and output pins. Here, the input pins `s1` and `s2` as well as the output pins `d1` and `d2` are explicitly passed to the called `Activity1` (l.2). An equivalent representation of the model is shown in Figure 10.3 at the bottom. It uses the auto-connect pin (l.1) such that the mapping of input and output pins can be neglected (ll.2-4). However, if the type auto-connect mode is chosen alternatively, this example will fail, because both input pins and output pins have the same type.

ADJava

```
1 action Action1(String s1, String s2) : Date d1, Date d2 {
2   execute dex.activities.Activity1(s1,s2).o1,o2 -> d1, d2;
3 }
```

ADJava

```
1 autoconnect pin;
2 action Action1(String s1, String s2) : Date d1, Date d2 {
3   execute dex.activities.Activity1;
4 }
```

Figure 10.3: The call behavior connector simplifies the definition of call behavior actions.

Activity Input Connector

The activity input connector allows to neglect mappings between the input pins of the activities and the input pins of an action. It implicitly maps input pins of an activity

to input pins of actions depending on the chosen auto-connect mode. However, this connector is only applied to actions that are explicitly connected to the initial node.

For example, an explicit mapping is shown in Figure 10.4 (at the top). It connects the input pin `i1` of the activity (l.1) to the input pin `i1` of the action `Action1` (l.2). This mapping is explicitly defined in l.3. At the bottom of Figure 10.4, an equal model is shown, where the mapping is neglected because the auto-connect pin mode is selected. Alternatively, the type auto-connect mode can be used in this example, because there is only one pin with the type `String`.

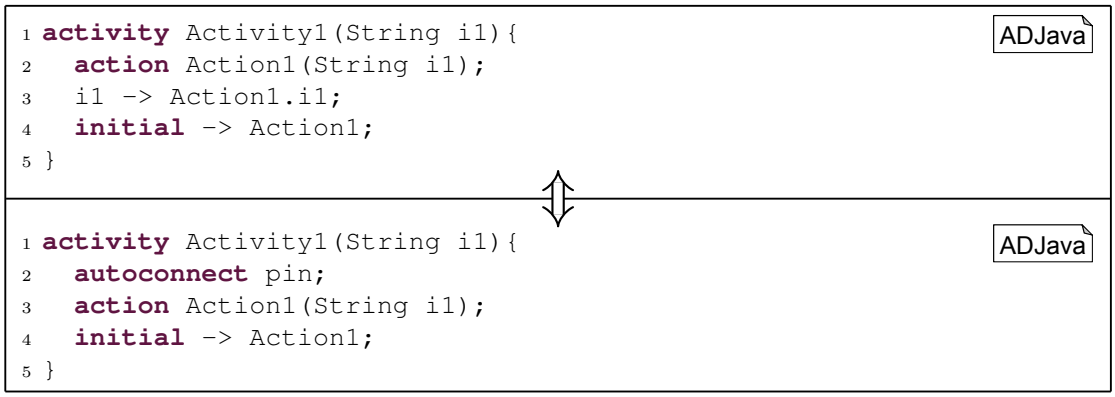


Figure 10.4: The activity input connector connects the activity's input pins to the action's input pins.

Action Output Connector

The action output connector simplifies the definition of object flows between actions. It connects all not explicitly mapped output pins of actions to input pins of actions that would otherwise have to be connected via object flows.

For example, the explicit definition of an object flow is shown in Figure 10.5 (at the top). The object flow between the `Action1` action (l.1) and the `Action2` action (l.2) is defined by mapping the `o1` output pin of the primer to the `i1` input pin of the latter action (l.3). An equivalent model is shown in Figure 10.5 (at the bottom), which uses the auto-connect type mode.

Control Node Connector

Each object flow can be directed through control nodes. For each such control node, the input and output pins have to be explicitly mapped. Hence, control type connector connects input and output pins of control nodes making object flows implicit.

```
1 action Action1 : String o1;                    ADJava
2 action Action2(String i1);
3 Action1.o1 -> Action2.i1;
```

```
1 autoconnect type;                              ADJava
2 action Action1 : String o1;
3 action Action2(String i1);
```

Figure 10.5: The action output connector connects output pins to input pins.

For example, Figure 10.6 (at the top) shows an example of an object flow between `Action1`, `Action2`, and `Action3` (ll.1-3), which is directed through a fork node (l.4). In this case, the output pin `o1` of `Action1` is mapped to the input pin `i1` of `Action2` and `i2` of `Action3` (l.4). To simplify this mapping, the auto-connect type mode can be used, as shown in Figure 10.6 (at the bottom). The defined control flow in l.4 (at the bottom) becomes an implicit object flow.

```
1 action Action1: String o1;                     ADJava
2 action Action2(String i1);
3 action Action3(String i2);
4 Action1.o1 -> Action2.i1 || Action3.i2;
```

```
1 autoconnect type;                              ADJava
2 action Action1: String o1;
3 action Action2(String i1);
4 action Action3(String i2);
5 Action1 -> Action2 || Action3;
```

Figure 10.6: Object flows through control nodes can either be defined explicit (at the top) or implicit (at the bottom) using the control node connector.

10.3 Execution of ADJava Models

To execute ADJava models, different approaches for execution of the UML AD ML, which forms the basis for the ADJava ML, exist. In particular, execution of UML AD models by code generation such as [NZ00, AT01, UN09, GR11, PADS12, Rei15, DKN+15]. An alternative approach is execution by interpretation, i.e., a generic interpreter executes UML AD models, such as [EW01a, EW01b, KG10, MLK12, LBG13]. However, code generation from ADJava models demands that a code generator has to be deployed with a data-centric application to allow application developers to redesign or create processes.

Interpretation of ADJava models requires just-in-time compilation of Java source code and reflective access to execute it, which is not targeted in this thesis (cf. Section 3.2 and Section 5.1).

Hence, an integrated interpretation- and code generation-based approach for execution of ADJava models has been developed, as shown in Figure 10.7. Execution of ADJava models is separated into an interpretation part (cf. *GR-4-2*) and an optional code generation part (cf. *GR-4-4*). In this approach, a modeler creates a CD4A model, which describes the data structure to be managed. Modelers also create different ADJava models describing processes that are intended to be supported in the generated data-centric application. Each ADJava model can use types defined in the CD4A. In addition, a ADJava model can be enrich with Java source code by a senior application developer to implement actions, guards, or conditions.

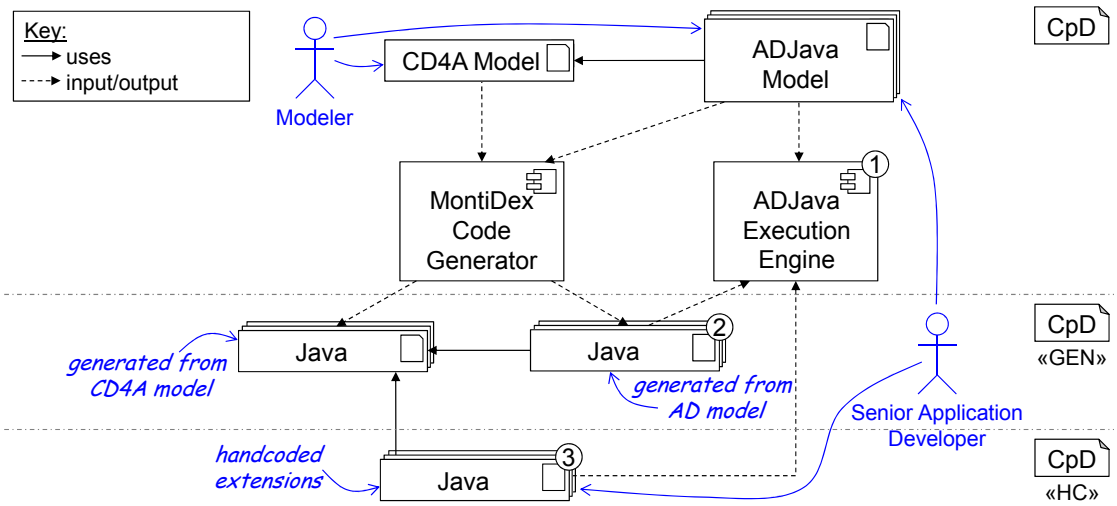


Figure 10.7: An overview of the interpretation- and code generation-based approach for ADJava model execution.

The execution of an ADJava model depends on the use of Java source code in the ADJava model. In particular, an ADJava model without embedded Java source code is executed by the `ADJava Execution Engine` (①) in Figure 10.7), which is an interpreter for ADJava models. Interpretation of ADJava models is explained in more detail in Section 10.3.1. Such ADJava models without Java source code can be extended with guard, condition, and action implementations by application developers using handcoded extensions of the `ADJava Execution Engine` (③), which are realized as described in Section 6.2 (cf. *GR-4-3*). To reduce the manual effort of creating handcoded extensions, application developers can enrich the ADJava model with Java source code. In this case, before executing the ADJava model using the `ADJava Execution Engine`,

code generation is required. In this code generation step, for every guard, condition, and action implementation, a Java source file is generated ((2)). Each generated Java source file contains the guard, condition, or action implementation that is executed during interpretation by the `ADJava Execution Engine`.

At interpretation of an ADJava model, the `ADJava Execution Engine` ensures that the correct handcoded extension and generated Java implementation for a particular guard, condition, and action implementation is executed. In particular, if code generation has been used to generate Java source code from ADJava models, the interpreter uses a *hash-based approach*, i.e., a hash code of the embedded Java source code of an action, a guard, or a condition in the ADJava model is computed and associated with the corresponding generate a Java implementation. If handcoded extensions of the interpreter are provided, an application developer has to manually register the manually-written Java implementation to the `ADJava Execution Engine` for a particular action by using the action's name (a detailed explanation is given in Section 10.3.3).

An approach of integrating interpretation and code generation for UML AD has already been proposed. In particular, an integration via a sequential execution of interpretation (called simulation) and code generation to enable validated code generation from UML AD models [BS05]. Moreover, Gessenharter presented the use of interpretation and code generation to increase the interpretation performance by generating source code containing additional information about the execution sequence [Ges10]. In addition, integration of handwritten code and interpretation on the model-level, i.e., integration of different DSLs via language extension mechanisms, and a compiler integration to reduce UML AD model complexity, which refers to the amount of used modeling elements, has already been successfully used [Sar06].

In the remainder of this section, first, the interpretation of ADJava models is explained. Afterwards, we describe how code generation benefits interpretation. Finally, we provide the technical realization and integration into MontiDEx products.

10.3.1 Method for Interpretation of ADJava Models

The interpretation part is rooted on the UML AD semantics, which are based on Petri Nets [Mur89] since UML 2.0 [www15b]. Activity diagrams are considered as a flow of tokens through a directed graph. Based on this understanding, different approaches have been proposed to execute UML AD models. Mayerhofer proposes a model execution framework [May14] that enables execution, debugging, testing, and validation of UML AD models based on fUML [www16h]. A similar approach based on fUML and focusing on debugging has been proposed [LBG13]. Another approach, which is used as the foundation for this thesis, proposes the ACTi interpreter using UML CD and UML AD in Activity Diagram Linear Form to execute activities and actions [CD08]. Finally, a

more general approach to support model execution of DSLs has been presented by the GEMOC Studio [BDV+16].

In this thesis, the following assumptions for executing ADJava models are made. First, parallel execution is omitted. Instead, it is resolved to sequential execution, which is managed by a scheduler, of actions and control nodes to avoid synchronization concerns. Second, it is assumed that interpretation is always possible but may raise a run-time exception, if the ADJava model contains Java source code that has not previously been processed in a code generation step before. Third, we do not aim to resolve semantic variations in the execution semantics, which may cause non-determinism for certain UML AD models (cf. [SF07, Ges10]). A proposed solution is to provide different execution strategies to resolve such non-determinism (*random* and *guided* in [CD08]).

An overview of the proposed approach is shown in Figure 10.8. A detailed description of the technical realization is provided in our previous work [LN16].

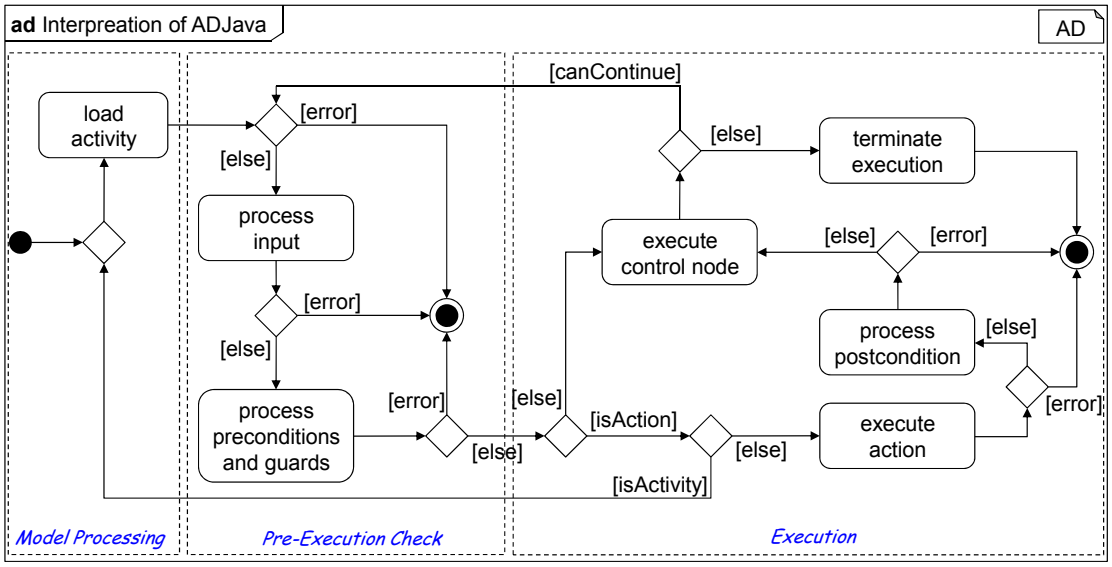


Figure 10.8: An abstract view of the process of interpreting an ADJava model consists of processing the ADJava model (Model Processing); setting input variables and checking preconditions and guards (Pre-Execution Check); and executing the action body and check post conditions (Execution).

Interpretation starts by processing an ADJava model (cf. Section 10.2). If during this language processing an error occurs, it is reported to the user and the execution terminates. Otherwise, the process to execute the model is started. In more detail, if the activity has input pins, the passed values are assigned. In general, this step requires code generation or handcoded extensions, because external data types to define the input pin's

type are used. Hence, it has to be ensured that the correct type of object is passed to the activity without use of reflection (cf. Section 5.1). This is ensured using the Double Dispatching-Pattern, as explained in Section 7.1.2. If input pins are defined, the interpreter uses the hash-based approach to locate the generated or manually-written source code and executes it. Technically, a hashmap is used to register Java objects instantiated from the generated Java source code, as it is explained in more detail in Section 10.3.3. For models without input pins, code generation or handcoded extensions are not required and this step is skipped. All errors during the processing of input pins lead to termination of the interpretation process.

After the input pins have been processed and their values have been assigned, preconditions and guards are checked. If a precondition or guard is defined, the hash-based approach is used to locate the generated or manually-written source code, which is then executed. If no generated or handcoded source code is found, execution fails. Execution also fails, if the executed precondition evaluates to false. However, if a guard fails, only the corresponding flow is dropped but the execution does not terminate.

After evaluating the precondition, the action or control node is executed. In case of a control node, the next elements to be processed are selected for the next execution cycle, which starts by processing the input pins. If the control node has to handle an object flow, the execution engine supports two different types to handle the objects. First, an object can be copied and each following node receives a copy. However, this may require merging data in join or merge nodes. Second, the data is not copied but handled as one instance that is passed to each following node.

If the node to be processed is not a control node and the action is not a call of an activity, then the action is executed, i.e., the action's body is executed, which is the generated or manually-written source code. If the action has no implementing body, the execution of the action is finished. Alternatively, if the action contains an activity call, the model processing step starts by loading the called activity, if it has not previously been called.

After an action has been executed, its postcondition is processed. This is done in same way as preconditions are handled, i.e., execution of handcoded extensions or generated Java source code, and, hence, may require code generation or handcoded extensions.

Finally, if not all elements of the model are processed, the interpretation continues. Otherwise, it is terminated. However, if an action called another activity, the overall execution terminates once the called activity has terminated.

10.3.2 Code Generation from ADJava Models

Whenever an ADJava model contains Java source code, code generation is used to generate executable Java source code, which uses the Double Dispatching-Pattern. In more

detail, for each ADJava, a dispatching interface containing a `visit()`-method for all used types in the ADJava is generated. Each method uses the Java `default` concept to realize a default implementation, which is empty to ensure that execution terminates if double dispatching fails. Since double dispatching is only applicable for non-primitive data types, wrapper types are used and extended with an additional `accept()`-method.

The example in Figure 10.9 shows an ADJava model (at the top left) that contains `Type1`, `Type2`, and `Type3`, each of which is defined the CD4A model (at the top right). For the ADJava model, the interface `ActivityObjectVisitor` (at the bottom) is generated. It contains a visit method for each type (ll.2-4). The second `String` parameter is required to pass the input/output pin's name.

ADJava

```
1 activity Activity {
2   data Type1 t1;
3   action Type2Action(Type2 t2);
4   action Type3Action(Type3 t3)
5 }
```

CD4A

```
1 classdiagram Types {
2   class Type1;
3   class Type2;
4   class Type3;
5 }
```

Java

```
1 public interface ActivityObjectVisitor
2                  extends ADObjectVisitor {
3   public default void visit(Type1 p, String n){ }
4   public default void visit(Type2 p, String n){ }
5   public default void visit(Type3 p, String n){ }
6 }
```

Figure 10.9: An example of the generated `visit()`-methods for each used data type to realize the Double Dispatching-Pattern.

The generated Java source code for a guard, a precondition, and a postcondition implements the dispatching interface whenever type information is required. In the remainder of this section, code generation from guards, conditions, and action implementations in ADJava models is described in more detail.

Generating Java Source Code from Input and Output Pins

If an ADJava model contains actions or defines an activity with input pins, Java source code is generated to implement the dispatching interface for all input pin types. For output pins no dispatching is required, because they are processed as input pins by the subsequent action or control node.

For example, Figure 10.10 shows the generated source code for the `CreateTransaction` action at the top. For this action the `CreateTransactionBody` Java class is generated. It extends the generic infrastructure provided by the execution engine

(AbstractAction in l.2 at the bottom) and implements the generated dispatching interface SubmitTransactionObjectVisitor (l.3 at the bottom). For each input pin and output pin, a protected variable with the defined type and name stores the values (l.4). In addition, an init()-method, which initializes the variables is generated. Furthermore, for the input pin i, the visit()-method having the type Customer is generated (ll.11-15 at the bottom), which dispatches the correct type and checks if the name is correct. For the output pin the getOutputs()-method is generated, which passes the result to the execution engine such that it forward it to the next input pin.

```
1 action CreateTransaction(Customer i): Customer o {
2   ...
3 }
```

ADJava TranSub
...

```
1 public class CreateTransactionBody
2              extends AbstractAction
3              implements SubmitTransactionObjectVisitor {
4   protected Customer i;
5
6   public void init() {
7     super.init();
8     this.i = null;
9   }
10
11  public void visit(Customer param, String name) {
12    if (name.equals("i")) {
13      this.i = param;
14    }
15  }
16 }
```

Java
«GEN»
...

Figure 10.10: The generated Java source code (at the bottom) generated from the ADJava model (at the top) to realize double dispatching for input pins.

Generating Java Source Code from Action Implementations

For each action in an ADJava model that contains an implementation, a Java class is generated. It contains the specified implementation in a doExecute()-method, which is executed by the execution engine. If the action defines additional input and output pins, the Double Dispatching-Pattern is added as described in Section 10.3.2.

For instance, the action in Figure 10.11 (at the top) shows an action implementation in the ADJava model. The generated Java class ExecutableJavaCodeBody (at the bottom) contains the specified Java source code in the doExecute()-method in ll.8-11.

```
1 action ExecutableJavaCode {
2   // Java source code
3 }
```

ADJava

```
1 public class ExecutableJavaCodeBody
2               extends AbstractAction
3               implements AObjectVisitor {
4   protected void doExecute() {
5     // Java source code
6   }
7 }
```

Java
«GEN»
...

Figure 10.11: For actions with implementations (at the top) a Java class with a `doExecute()`-method (ll.8-11) is generated.

Generating Java Source Code from Pre- and Postconditions

If an ADJava model contains preconditions or postconditions, a Java source file is generated for each precondition and postcondition. It implements the dispatching interface to realize double dispatching. Additionally, it contains the `doEvaluate()`-method, which contains the precondition specified in the ADJava model. Pre- and postcondition implementations are generated into separate artifacts. For instance, for the action `Action1` with a precondition and a postcondition, the artifacts `Action1Precondition` and `Action1Postcondition` are generated.

For example, the action in Figure 10.12 (at the top) specifies a precondition. For this action, the `ValidateTransactionPrecondition` Java file (at the bottom) is generated. It contains the input pin (l.4) and implements the dispatching interface (ll.11-15). The specified precondition is defined in the `doEvaluate()`-method in ll.17-19.

```
1 [t != null]
2 action ValidateTransaction(Transaction t);
```

ADJava TranSub
...

```
1 public class ValidateTransactionPrecondition
2               extends AbstractCondition
3               implements SubmitTransactionVisitor {
4   protected boolean doEvaluate() {
5     return t != null;
6   }
7 }
```

Java
«GEN»
...

Figure 10.12: The precondition defined for the `ValidateTransaction` action (at the top) is generated to Java source code (at the bottom).

10.3.3 Technical Realization of the Extended Data-Centric Infrastructure

The data-centric infrastructure presented in Chapter 7 has been extended to realize the proposed approach to execute ADJava models (cf. Section 10.3.1). In particular, the generated presentation layer has been extended with an additional view as shown in Figure 3.4 on page 31. It provides functionality to select, run, pause, stop, and save the state of an ADJava model. Moreover, the execution engine is part of the RTE (cf. Section 7.1.3). Hence, it is deployed with each MontiDEx product.

The main elements of the technical realization are shown in Figure 10.13. It depicts a simplified view on the implementation to demonstrate the main elements. The full technical realization can be found in [LN16].

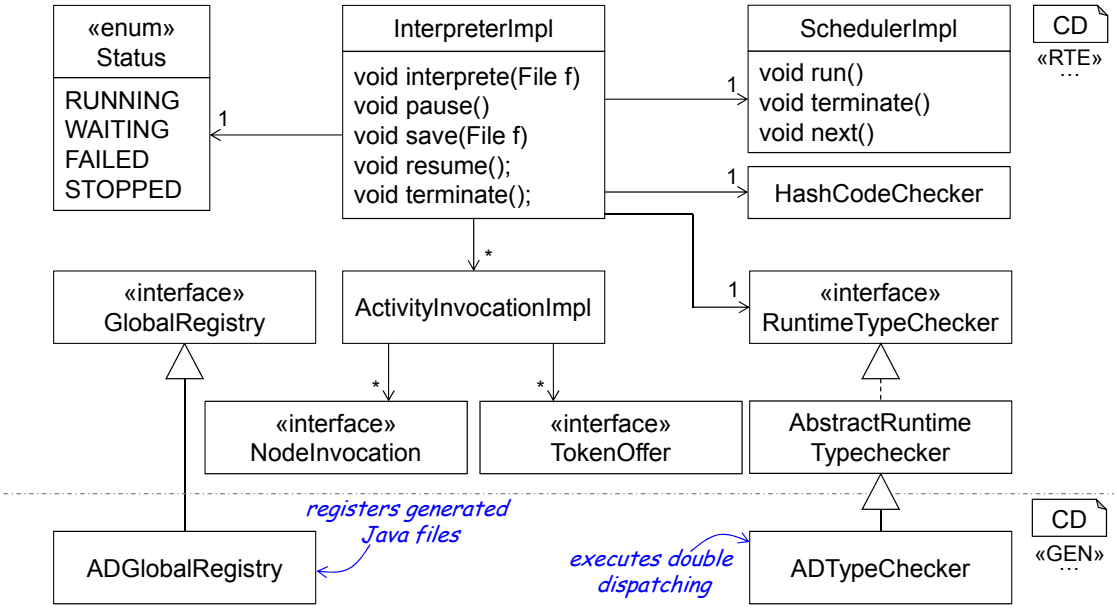


Figure 10.13: Overview of the technical realization of the AD Execution Engine.

The execution engine is realized in the `InterpreterImpl`, which provides the following methods:

- **`interprete(File f)`:** Start the execution of the UML AD model `f`. Each valid model is immediately processed and the its interpretation starts as described in Section 10.3.1.
- **`pause()`:** Interrupt execution if the interpretation has previously been started. The current state is not serialized but kept in memory only.

- **`save(File f):`** Serialize the current state of the execution into a file and can be loaded to continue execution.
- **`resume():`** Resume execution if it has previously been paused.
- **`terminate():`** Terminates execution immediately. This does not terminate execution of action implementations.

Each execution of an ADJava model starts by instantiating an `ActivityInvocationImpl`, which represents the activity currently executed. Because actions can call other activities, each `InterpreterImpl` instance manages all currently executed activities. Each activity manages its own actions and control nodes that are currently executed, which is represented by a `NodeInvocation` instance. For presentational reasons the concrete subclasses for each action and control node are omitted. Furthermore, it manages tokens, which represent markers on the currently executed nodes, in a queue. Hence, each `TokenOffer` represents a currently active token that has to be processed.

Technically, the execution engine is realized using a scheduler, which has to perform the following steps (based on [CD08]):

- Selects the actions or control nodes to be processed on round-trip basis.
- Execute the actions or control nodes.
- Pass tokens to the subsequent actions or control nodes.

The scheduler can be paused before and after processing a model element, i.e., an action implementation cannot be paused. Moreover, it supports stepwise execution to enable debugging (cf. *GR-4-1*). When the execution is started, the scheduler is started by creating an `SchedulerImpl` instance. It allows to run, terminate, and execute the next step. Execution is automatically terminated by the scheduler, if there are no more tokens to be processed.

In addition, the `HashCodeChecker` is the technical realization of the hash-based approach to identify generated (cf. Section 10.3).

Besides the run-time implementation, for each ADJava model a subclass of the `GlobalRegistry` and the `AbstractRuntimeTypeChecker` is generated. In the example in Figure 10.13, the `ADGlobalRegistry` is generated to store all generated and manually-written files registered for a particular hash code, as explained in Section 10.3. The `ADTypeChecker` contains the implementation of Double Dispatching-Pattern for each data type in the ADJava model as described in Section 10.3.2.

10.3.4 Technical Realization of the MontiDEx Code Generator Extension

The MontiDEx code generator has been extended with a default configuration as shown in Figure 10.14. It consists of a Java class (`AdScript` class) that enables support for

processing ADJava models (`parseAD()`-method) and checking ADJava model context conditions (`checkADCoCos()`-method). In addition, the `createSymbolTable()`-method allows to create a symbol table for a parsed UML AD model.

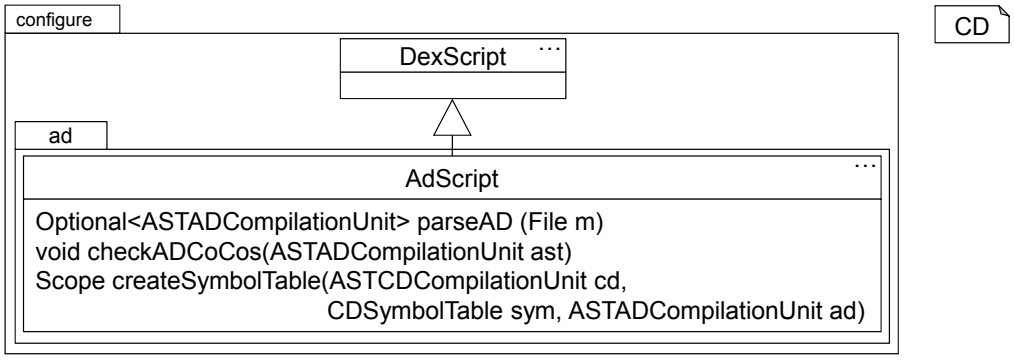


Figure 10.14: MontiDEx code generator default configuration to process ADJava models.

Besides a default configuration, an activity MontiDEx module in the MontiDEx code generator is provided, as shown in Figure 10.15. For presentational reasons, only the transformations are illustrated and templates are omitted. The `ADObject` as well as the `ConcreteADObjectVisitor` classes generate the Double Dispatching-Pattern. In particular, the `ADObject` transformation adds the `accept()`-methods and the `ConcreteADObjectVisitor` adds the `visit()`-methods. The `ConcreteTypeChecker` class adds the interface containing a method for each type used in the activity that is required to be implemented (cf. Section 10.3.2). The `ConcreteObjectConverter` adds serialization capabilities (cf. *GR-5-2*). Finally, the `ActionBody` and `Condition` transformations realize the code generation for action bodies, conditions and guards.

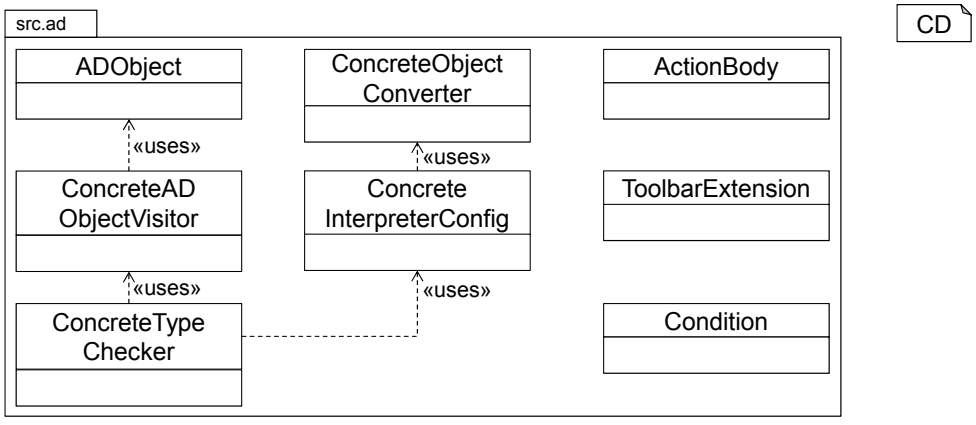


Figure 10.15: Transformations added to extend the MontiDex code generator.

10.4 Method for Developing Processes with ADJava

To support process automation of data-centric applications, this section presents a method to use the developed concepts for ADJava modeling and execution. The method shown in Figure 10.16 guides modelers and (senior) application developers in developing and extending ADJava models. Furthermore, it helps the decision on when to use hand-coded extensions or embedded Java in ADJava models to implement actions, guards, and conditions.

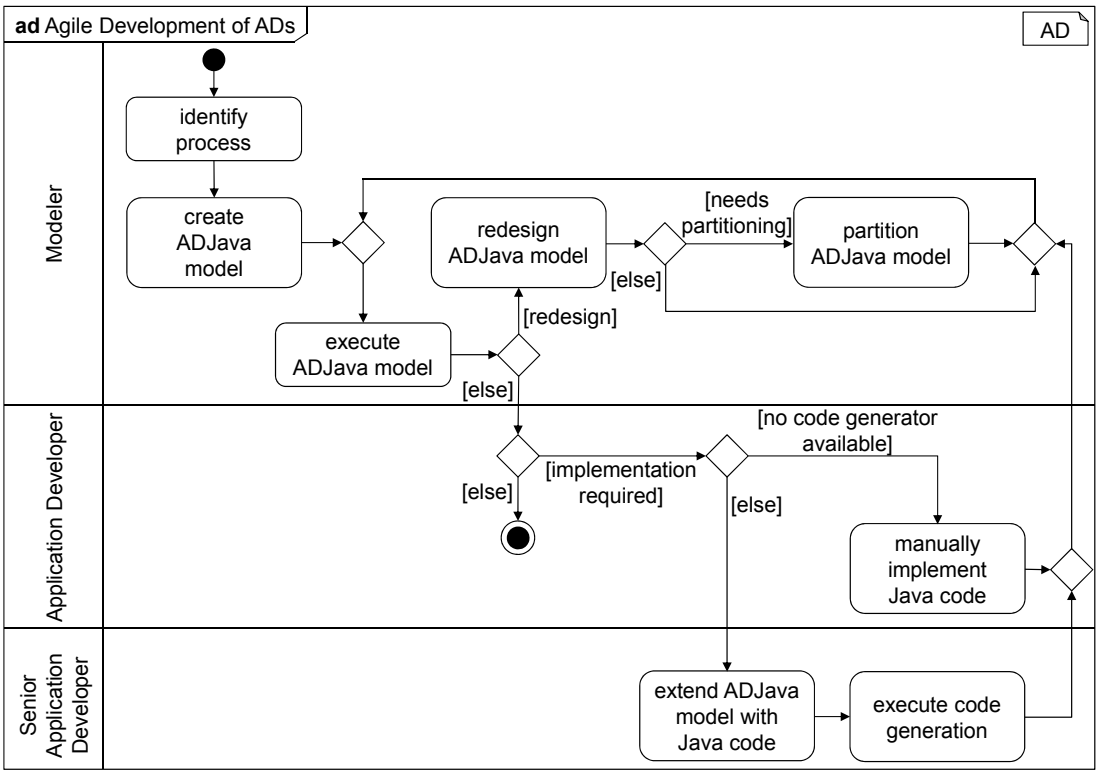


Figure 10.16: An overview of the process of modeling and executing of ADJava models.

The first step is to identify the process that is to be modeled. This task is done by an modeler during requirements analysis. If multiple processes are identified, the subsequent steps have to be applied individually to each process.

Afterwards, the modeler creates an ADJava model using the ADJava ML. This ADJava model does not contain any guard, condition, or action implementation. Such prototypical ADJava models can be executed to simulate the process. If any redesign attempts are identified during process simulation, e.g., missing actions, then the ADJava

model is redesigned by the modeler. In addition, a modeler can also partition an ADJava model into multiple activities and reference each other using the call behavior action. A partitioning is necessary in the following cases: reduce complexity of processes, i.e., actions necessary to define the process; extract a self-contained part of the process for reuse purposes.

If there are no redesign attempts, guard, conditions, and action implementations can be added by the application developer. This can either be done by using the supported embedded Java within the ADJava ML, which requires code generation to generate executable Java source code. However, this has to be done by a senior application developer. Alternatively, if the code generator is not available handcoded extensions can be added to the interpreter. Such handcoded extensions have to be manually registered in the interpreter (cf. Section 10.3.3). Afterwards, the ADJava model can be executed.

After implementing guards, conditions, and actions, the ADJava model can be executed. Redesign of the ADJava model and handcoded extensions can always be performed. This cycle of process automation development is finished if the process identified by the modeler is fully realized.

10.5 Evaluation and Limitation

This chapter summarizes the customization and adaptation mechanisms used to realize the extended infrastructure for process automation. Furthermore, it discusses limitations of the proposed approach. We are aware that the given lines of code (LoC) and implementation only reflects our technical realization and are not statistically sound. However, it gives a first indication to a potential reduced development time.

10.5.1 Evaluation of MontiDEx Customization and Adaptation Approaches

An overview of all chosen customization and adaptation mechanisms used to realize the extended infrastructure is shown in Table 10.9. Subsequently, each of them is explained.

The presented extended infrastructure uses the ADJava ML, which has been developed with MC, for process modeling. To process ADJava models, the MontiDEx code generator has been extended with a default configuration script, as described in Section 10.3.4, by using the proposed approach presented in Section 9.3.1. It consists of a Groovy script (231 LoC) and a Java configuration class (210 LoC).

Each generated MontiDEx product requires an ADJava Execution engine to execute ADJava models. Hence, each MontiDEx product has to be deployed with the ADJava Execution engine. Technically, it is realized as a second RTE, as shown in Figure 3.5 on page 39. Hence, it does not require any adaptation of the MontiDEx.

Handcoded Extensions Approach	Used	Unused
Extended Generation Gap		
- Signature Extension		✓
- Implementation Extension		✓
- Hot Spots		✓
Template Attachments		
- Add before		✓
- Add after		✓
- Replace	✓	
Template Extensions		
- Add before		✓
- Add after		✓
- Replace	✓	
Template Hook Point	✓	

Table 10.9: The customization and adaptation approaches used to realize the extended infrastructure for process automation.

To realize the code generation from ADJava models, as described in Section 10.3.2, one MontiDEx module has been developed. It consists of nine templates (182 LoC) and eight transformations (769 LoC) as shown in Section 10.3.4. The technical realization uses the common infrastructure to realize MontiDEx modules, which is introduced in Section 9.2.

Since ADJava models support CD4A types in object flows, the generated source code has been extended such that the ADJava Execution Engine can process CD4A types. Therefore, each generated CD4A class extends the `ADObject` interface, which realizes double dispatching. Technically, this is realized using a template extension, which replaces the default template and adds a modified one. In addition, the `Concrete-Controller:addToolBarButton` hook point has been used to extend the GUI to provide support for ADJava execution, as described in Section 3.3.2. This GUI required a minimal extension of the RTE to provide the static non-generated source code.

In summary, the MontiDEx code generator and product could be extended with the provided customization and adaptation mechanisms. However, additional hook points had to be added to realize the required functionality. Therefore, the provided methods have shown to be valuable assets to support unexperienced users to adapt and extend the code generator.

10.5.2 Limitations

Due to the requirements (cf. Section 3.2), design guidelines (cf. Section 5.1), and targeted development methods (cf. Section 3.3), the interpretation- and code generation-based

approach for ADJava model execution harbors limitations. In the remainder of this section, these limitations are explained.

The first limitation is the need to deploy the interpreter with the MontiDEx product. In particular, to execute an ADJava model, each generated MontiDEx product has to be deployed with the interpreter contained. This introduces additional unnecessary dependencies, because the interpreter itself requires the ADJava ML. In addition to the deployment of the interpreter, the code generator has to be deployed as well, if implementation concerns are realized in the ADJava model and handcoded extensions of the interpreter are not wanted. For handcoded extensions, the MontiDEx product's source code is required to compile the manually-written Java source code.

Another restriction arises from the client-side ADJava model execution. In this case, the roles modeled cannot be used for collaborative working, because the process is executed by one particular end user and the state is stored locally. Hence, to support such scenarios, it has to be executed on server-side. However, such a solution yields additional challenges, e.g., persistence of different states of execution among different end users.

A further restriction is based on the ADJava ML supports of auto-connect capabilities, and control and object flows chaining. In particular, auto-connect may result in unwanted side-effects during model execution. The reason is that the implicit mapping of input and output pins (cf. Section 10.2.6). Such implicit mapping form control and object flows, which are not visible a priori but only when executing the ADJava model. Hence, modelers should avoid the same name and type of input pins for different actions, activities, and data types if auto-connect is not explicitly wanted. As a consequence, the presented approach enforces more responsibility to the modeler because he has to ensure that the model conforms to the semantics (cf. [CD08]), i.e., no semantic variations are introduced.

Finally, the source code generated from an ADJava model ensures type-safety without reflective access but uses `String` comparisons to connect input and output pins. This is, however, necessary to ensure that auto-connect pin capabilities are supported.

Chapter 11

Case Example: MDP and MDD with MontiDEx

The concepts, methods, and tools developed in this thesis have been used in the development of multiple data-centric applications and data-centric application prototypes. The MontiDEx product (cf. Chapter 9) has been used standalone, i.e., including all layers of the data-centric infrastructure, as well as a framework-like part of a software system, i.e., only parts of the data-centric infrastructure have been used.

In this chapter, case examples developed in the course of this thesis are presented. For each case example the LoC generated, which alternatively would have to be manually-written, and LoC additionally required to be manually-written to implement the case example are compared. In addition, an overview of customization and adaptation approach used in each case example is given to demonstrate their use.

In general, such comparison is difficult (cf. [FRS13]). However, it provides an indication for a potential reduced development time, because the generated functionality is reused and does not have to be manually-written. Nonetheless, the collected data is not statistically sound, but only reflects the technical realization of each case example.

This chapter is structured as follows. First, a Points-of-Interest (POI) management system, which demonstrates framework-like use, is presented as the first case example in Section 11.1. The second case example for framework-like use is an audio and video streaming platform described in Section 11.2. Finally, MDP of an examination regulation system for universities is shown in Section 11.3 as an example for standalone use of the MontiDEx product.

11.1 Points-of-Interest Management System

The first case example is a POI management system, which manages radar traps, traffic jams, construction sites, accidents, and general POIs. An Android [www16f] application allows to manage POIs, which are stored with their geographical location and an optional

photo or a textual description. Also, it tracks the user's current position and displays nearby POIs, which are periodically updated and can also be filtered or ranked.

The UML CD model describing the data structure is shown in Figure 11.1. The CD4A model of this UML CD is presented in Listing D.2. It contains the `POI` class, which represents a general POI having a geographical location (`longitude` and `latitude` attributes), the creation date (`created` attribute), an optional description (`description` attribute), an optional photo (`photopath` attribute), and an optional deprecation date (`destroyed` attribute). Each `POI` has exactly one `POIType` and a particular `POIRating`, which represents the rating for this POI.

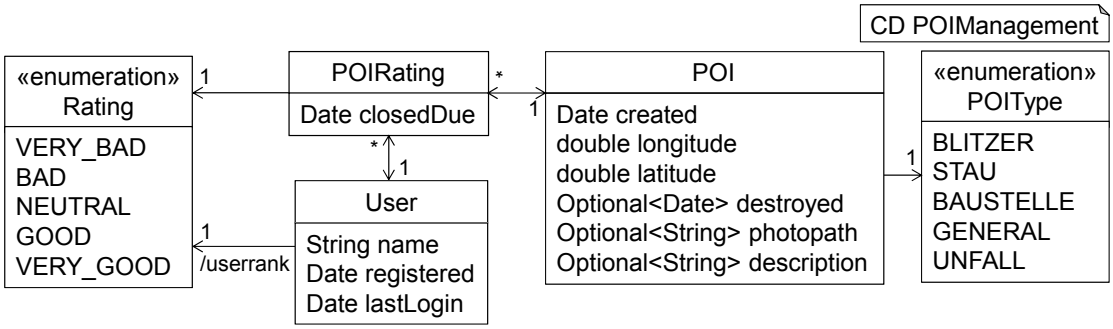


Figure 11.1: A UML CD for describing a POI management system.

Users are explicitly modeled (`User` class), because they are allowed to create ratings and can as well have a rating and a user rank (`userrank`) that represents their credibility. The rating and a POI's type is used to create a POI's expiration date. The better a POI is rated, the longer the POI is active, i.e., stored and displayed to users.

11.1.1 Technical Realization

This case example was realized within four months by two computer science students in the course of a software lab at the RWTH Aachen university. Both students were graduate students and experienced in Java programing. Because the primary goal of MontiDEx products are lightweight client applications, only the MontiDEx product application layer (cf. Section 7.2) and persistence layer (cf. Section 7.5) have been generated and used. With this generated part, the following extensions that have to be manually added have been identified:

- A manually-written native Android application allowing user interaction and connects to a RESTful service[1].

[1]Representational state transfer (REST) is an architectural style for design of web services [Fie00].

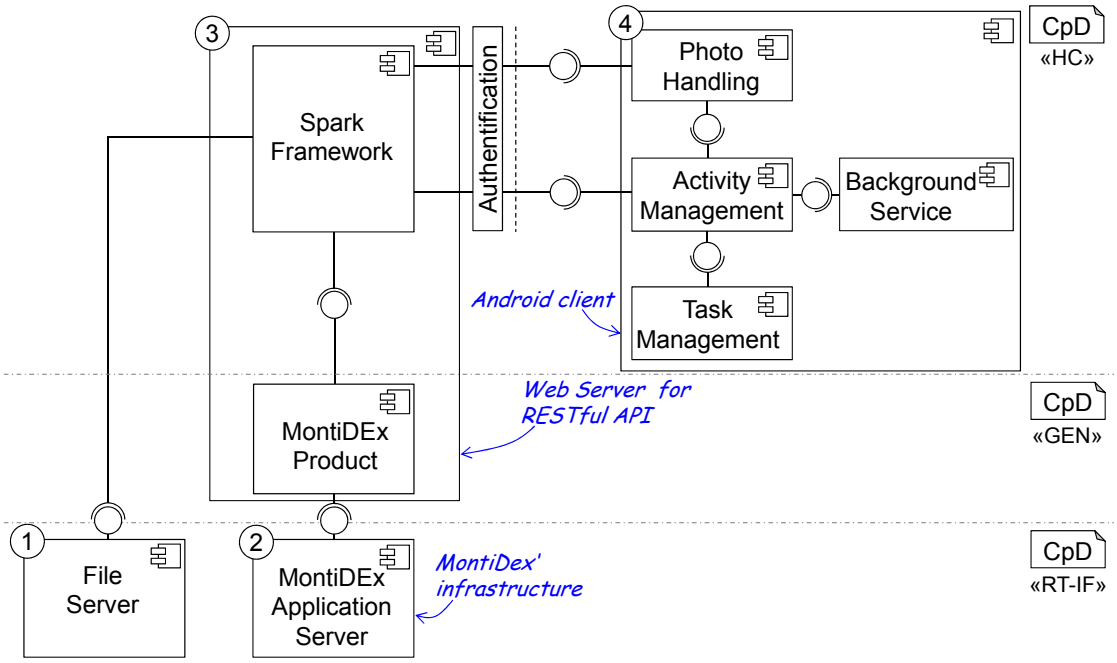


Figure 11.2: An overview of the POI Management system's client architecture consisting of an Android client, a Web Server, and the MontiDEx infrastructure.

- A web server providing RESTful services and running the MontiDEx product.
- An additional file storage support to store uploaded photos.

To fulfill these requirements, the architecture shown in Figure 11.2 has been developed. It consists of a file server to store the uploaded photos (①), the MontiDEx application server to store created instances of the data structure (②), a web server providing the RESTful service and running the headless MontiDEx product (③), and the Android client handling user interactions (④).

The MontiDEx application server stores instances of the modeled data structure only. To enable management of files, it contains the path to the uploaded file managed by the file server, i.e., the `photopath` attribute in the `POI` class in Figure 11.1. The path is a valid path on the file server. For each requested photo, the web server requests the stored file identified by the path from the file server and returns the result to the client.

The headless MontiDEx product, i.e., no presentation layer, is executed on a web server using the SparkJava framework [www16m] to provide a RESTful interface for the Android client (③ in Figure 11.2). It allows authenticated users to perform CRUD operations on the managed data. Technically, MontiDEx's user management system

(cf. Section 7.4.2) was reused and extended to provide the required REST authentication.

Besides authentication, the REST service offers the following functionality:

- Retrieve all POIs in a radius of less than 10 km.
- By explicitly stating the POI-ID a particular POI can be queried, i.e., that all its contained data is returned.
- Retrieve all POIs in a greater radius (20 km) to enable offline mode.
- Given a particular POI-ID, a rating can be set for a POI. Ratings can only be set once by a user.
- Create a new POI for a particular location. Optionally, attach an image.

These RESTful services are used by the Android client, which contains a `Photo-Handling` component allowing upload and download of Base64-encoded photos. Moreover, to handle user events, the front-end is subdivided into multiple Android activities, each of which handles one event, e.g., store POI. Those that require a connection are realized as asynchronous activities to avoid a blocking UI. All synchronous activities are managed by the `ActivityManagement` component, whereas all asynchronous activities are handled by the `TaskManagement` component.

The `BackgroundService` component allows to update the application's POI list without the need to execute the Android UI. When the client application is launched a background service is started to determine the user's location. If a connection to the server is established, the location is transmitted and all relevant POIs are retrieved, each of which is displayed in an overview as shown in Figure 11.3 (left side). If the user's current position changes by 1 km, this list is updated. In addition, a filter for the POI list allows to search for POI and a settings dialog is provided (right side).

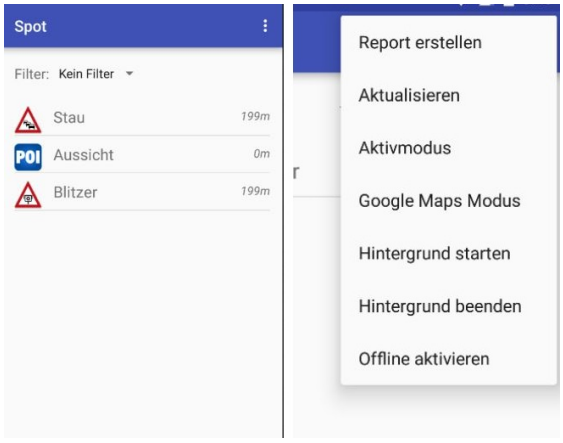


Figure 11.3: Android client's main UI (left) and configuration dialog (right).

11.1.2 Discussion

This case example presents the development of a management system using only the application and persistence layer of a MontiDEx product. It also shows the framework-like use of these parts as a back-end of a web server. This has been beneficial because the role-based-access control (cf. Section 7.5) has been reused to provide authentication for a REST interface to manage the different clients.

In general, this reuse of generated functionality indicates a potential speedup in the development time as shown in Table 11.1, which presents the LoCs of the generated and manually-written code excluding empty lines and comments. It shows that the Android client is handcoded, because the MontiDEx product targets lightweight client applications. Likewise, the SparkJava web server is manually-written. However, the MontiDEx product provided 99.87 % of the required management functionality, which is generated from 37 LoCs of the CD4A model (cf. Listing D.2), with only 0.13 % of handcoded extensions required to implement the derived association. In total, the overall application contains 89.62 % of generated code and 10.38 % of manually-written code. This equals a factor of 8.64 of generated to manually-written code.

	LoC		**Percentage**	
	handcoded	generated	handcoded	generated
Android Application	2119	0	100 %	0 %
SparkJava WebServer	720	0	100 %	0 %
MontiDEx Product	32	24799	0.13 %	99.87 %
Sum	**2871**	**24799**	**10.38** %	**89.62** %

Table 11.1: An overview of the LoC generated and manually-written for the POI management system.

Besides a LoC comparison, Table 11.2 gives an overview of the used customization and adaptation mechanisms provided by the MontiDEx product and code generator. It shows that the generated MontiDEx product is extended using the implementation extension. This is necessary to implement the derived association (`userrank` in Figure 11.1). Moreover, the replace template attachment operation is used by the MontiDEx code generator to replace the default templates (cf. Section 8.2.5).

11.2 Audio and Video Streaming Platform

The second case example is an audio and video streaming platform. It provides a Web-Front-end to manage different types of media, each of which can be viewed and rated

Handcoded Extensions Approach	Used	Unused
Extended Generation Gap		
- Signature Extension		✓
- Implementation Extension	✓	
- Hot Spots		✓
Template Attachments		
- Add before		✓
- Add after		✓
- Replace	✓	
Template Extensions		
- Add before		✓
- Add after		✓
- Replace		✓
Template Hook Point		✓

Table 11.2: Overview of used customization approaches for the POI management system.

by users but only uploaded by an administrator. In addition, functionality to manage watch lists, histories and comments is provided.

An overview of the data structure to be managed is shown in the UML CD model in Figure 11.4. Note that the corresponding CD4A model is shown in Listing D.3. It can be partitioned into user management with different profiles, media management including containers, rating and commenting functionality, watch lists, and history functionality.

Users are explicitly modeled (`Account` class in Figure 11.4) and associated with one `Profile` and one `Role`. While the `Account` class is a wrapper for the MontiDEx user management (cf. Section 7.4.2) to explicify the two supported roles `ADMIN` and `USER` (`Role` enumeration), a `Profile` is the user displayed in the application. This explicit modeling of users enables user management capabilities in the web-front-end.

A media file is represented by the `DexFile` class shown in Figure 11.4. It is either a file (`DexFileElement`) or a container (`DexFileContainer`), which groups files to represent, e.g., albums or series. Both store the amount of times they have been viewed in the `consumed` attribute. However, the container has a derived `consumed` attribute, because it is computed based on the consumption of the enclosed files. Moreover, for each `DexFile` additional meta information in the `DexFileMetaData` is stored such as age restriction (`FSK` enumeration), public availability (`Visibility` enumeration), and the file format (`MediaType` enumeration).

Rating and commenting functionality is offered by the `Rating` and `Comment` classes. Users can create multiple ratings and comments for a particular `DexFile` only once. Each comment receives a timestamp (`creationDate` attribute in the `Comment` class).

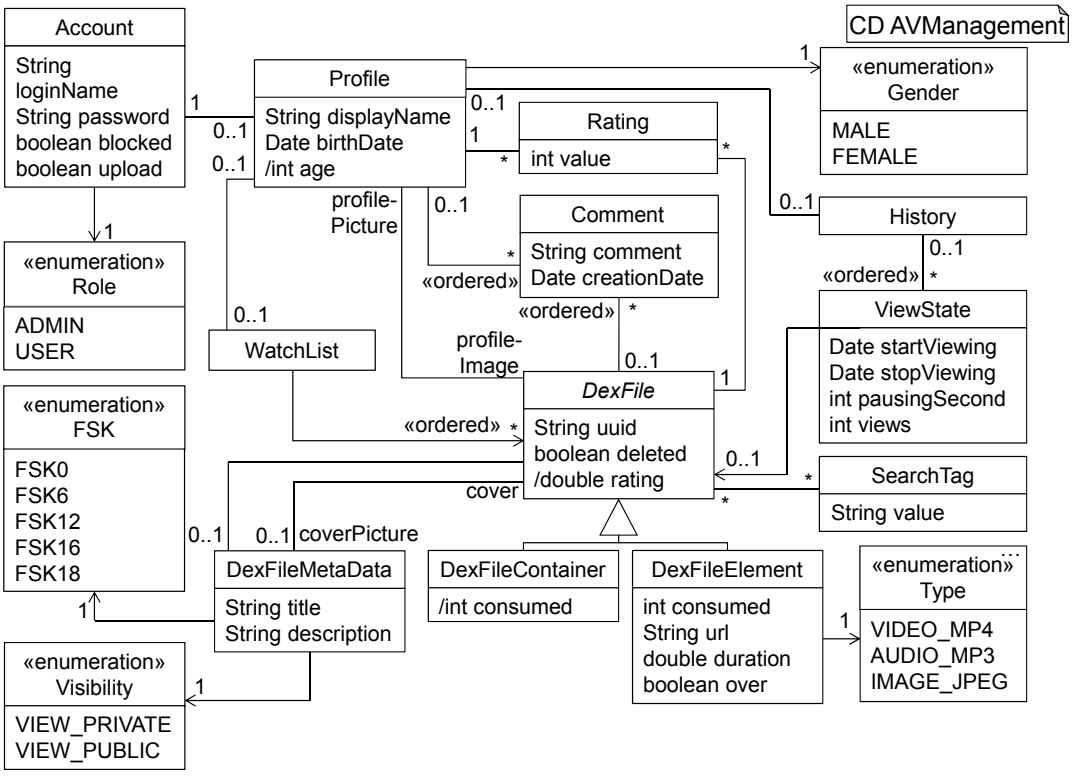


Figure 11.4: A UML CD showing the main elements to manage audio and video media.

Besides commenting and rating media, a user is allowed to mark media to be watched later, which is stored in a watch list (`WatchList` class). A watch list is modeled as an explicit class, because it is possible to share them among users.

Finally, history functionality is concerned with keeping track of the watched media for one particular user. Hence, each user is associated with the `History` class, which manages multiple `ViewStates` of watched media. A `ViewState` stores the time when the user has started (`startViewing` attribute) to watch the media as well as the date when the user finished watching (`stopViewing` attribute). In addition, the amount of views (`views` attribute) and the progress (`pausingSecond` attribute), i.e., time of the media that has been watched, is stored. This enables users to continue watching media at any the paused time.

11.2.1 Technical Realization

The technical realization of this case example was realized within four months by two computer science students in the course of a software lab at the RWTH Aachen univer-

sity. Both students were graduate students and highly experienced in web development. Before presenting the technical realization, we summarize the requirements for the targeted application as follows:

- Provide a web-front-end to view, rate, and comment media.
- Provide a GUI to manage users.
- Provide functionality and an infrastructure to upload media.
- Provide an infrastructure that manages the Web-Front-end's requests and gives access to the managed media.

The technical realization can be partitioned into a back-end, which is based on MontiDEx product to provide the necessary management functionality extended by a SeaweedFS [www16l] file server to store the uploaded files, and a front-end, which is based on the AngularJS [www16b] and the Spring Boot framework [www16n].

Back-end

The back-end architecture is shown in Figure 11.5. It provides RESTful services to clients allowing to manage the data structure and the uploaded files. Furthermore, it consists of the MontiDEx application server (②) for the data structure, a file server (①), which stores the media files, and a web server (③) that executes the headless MontiDEx product and provides a REST interface.

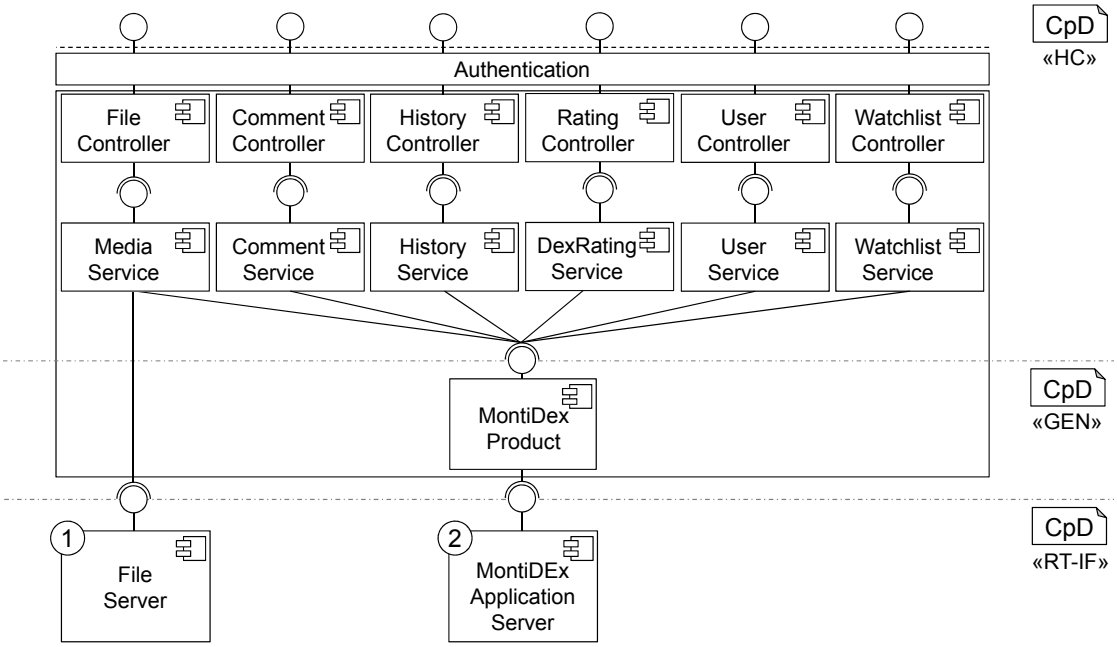


Figure 11.5: Overview of the back-end architecture.

The MontiDEx application server and the file server work in concert using the same integration approach as described in Section 11.1.1. Namely, using an attribute (`url` in the `DexFileElement` class in Figure 11.4) to store the path to the corresponding file.

To use the provided web services, a user has to be authenticated. Afterwards, the following controllers can be accessed, each of which provides on of the following REST services (Note that due to presentational reasons only the main services are listed.):

File Controller This service allows to upload and stream media files. A user can upload a media file and automatically receives the management rights for this particular file. The upload process works as follows:

1. Ensure that the uploaded file is supported. This is achieved by using the Apache Tika framework [www16e], which also supports prevention against uploading executable files.
2. Compute the length of the media file.
3. Send the file to the SeaweedFS distributed filesystem to store it permanently.
4. Create the corresponding media representation (`DexFileElement` object) in the MontiDEx database and store the path of the uploaded media file.

Comment Controller Manage comments for all stored files and containers.

History Controller Each valid user has a history in the system, which can be managed via this service. It contains information such as the upload of a particular file and when the last log in has taken place.

Rating Controller This interface can be used to manage the rating system. The rating is on a scale from 1 (very bad) to 5 (excellent).

User Controller This service allows to manage the different users and to distinguish between administrator and normal user. An administrator can view and manage all user accounts, while a user cannot. Each user - regardless of their role - can define their profile with a password, an avatar, and details. It is also possible to delete user accounts using this service.

Watchlist Controller A watch list is a collection of files that are marked as to be watched by a user. This service allows to manage such watch lists.

Front-End

The services provided by the back-end are used by dedicated services in the front-end, each of which is responsible for managing one particular back-end functionality and is realized by a dedicated controller. Each controller is responsible for directing user interactions to the services. It is also used to provide requested service results, as shown

in Figure 11.6. In addition, the front-end provides an authentication service to manage authentication concerns with the back-end and uses the Interceptor-Pattern [GHJV95] to allow interceptions during authentication. It also provides logging functionality and session management. Note that a detailed description of each controller and service is omitted, because they realized the functionality shown in Figure 11.5.

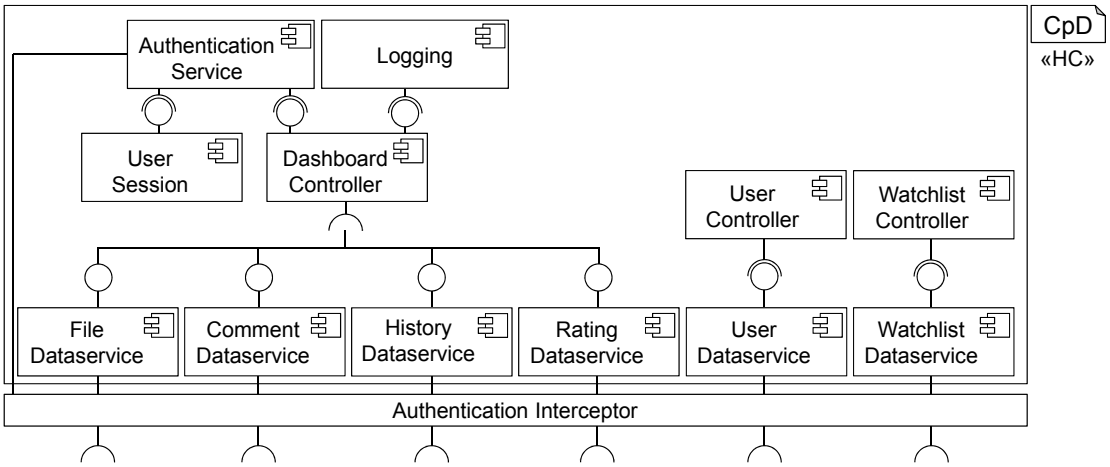


Figure 11.6: Overview of the front-end architecture.

The handcoded GUI provides multiple user interfaces for the varying concerns including a dashboard, user management, container management, search, watch list, and statistics overview. An example is illustrated in Figure 11.7. It shows all available files and containers. If a container or a file has a cover, then the cover is shown as well. The same applies to ratings and number of views for containers and files. Moreover, the dashboard allows to filter (by type, rating, and views) and search for files and containers.

Another example is the view for a selected media file, which is shown in Figure 11.8. Depending on the media file's type either audio or video is played. The overview shows a short description, the user who uploaded the media file, comments, and ratings. Note that only logged in users are allowed to create new comments.

11.2.2 Discussion

The demonstrated case example has been developed using the proposed lightweight method for MDD of data-centric applications (cf. Section 3.3.2). The overall results show a potential decrease of LoC to be written, when developing such software systems. In particular, because a Web-Front-end has been required, it has been manually-written to 100 %, as shown in Table 11.3. The same holds for the web server providing the REST interface. However, it uses the headless MontiDEx product, which consists of 0.14 %

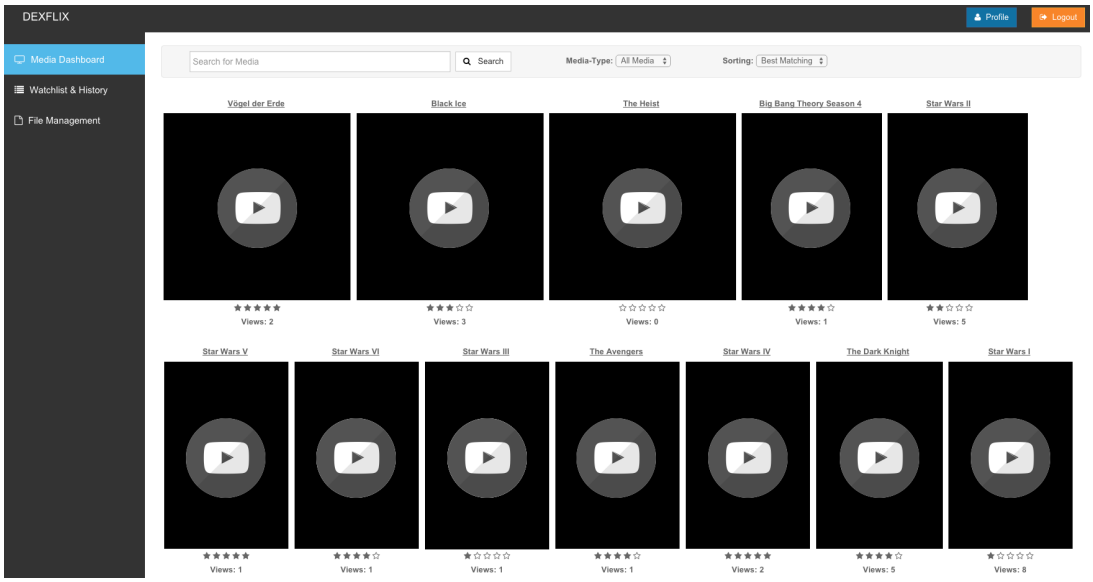


Figure 11.7: A screenshot of the dashboard showing containers and media files.

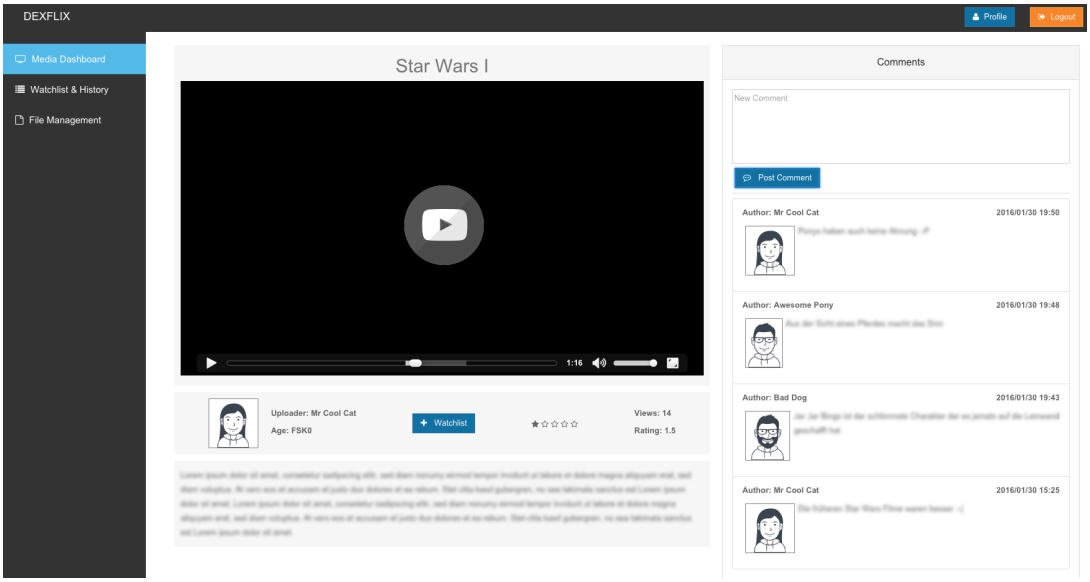


Figure 11.8: A screenshot of the view for playing video files.

manually-written LoC to implement the derived attributes and 99.86 % generated LoC, as shown in Figure 11.4. Note that the manually-written code demonstrates how to implement possible constraints such as a rating of 1 to 5 modeled as an `int` value. In total, the created software system consists of 7.87 % manually-written LoC and 92.13 % generated LoC, which represents a factor of 11.7.

	LoC		**Percentage**	
	handcoded	generated	handcoded	generated
AndularJS Web-Front-end	4458	0	100 %	0 %
Web server	5022	0	100 %	0 %
MontiDEx Product	155	112743	0.14 %	99.86 %
Sum	**9635**	**112743**	**7.87** %	**92.13** %

Table 11.3: Overview of the manually-written and generated LoC.

This example also shows the simultaneous use of a MontiDEx database server and a distributed file system. Moreover, it uses one client for user management and one for media management showing the use of different front-ends with the same back-end.

The manually-written code is realized using the Extended Generation Gap-Pattern (cf. Section 6.2). In this example, both the interface and the implementation extensions are used. In particular, interface extensions are used to add technical details, which have to be globally accessible. Moreover, only the template attachment replace operation are used by the MontiDEx code generator to generate the overall software system. An overview of all used customization approaches is shown in Table 11.4.

Handcoded Extensions Approach	**Used**	**Unused**
Extended Generation Gap		
- Signature Extension	✓	
- Implementation Extension	✓	
- Hot Spots	✓	
Template Attachments		
- Add before		✓
- Add after		✓
- Replace	✓	
Template Extensions		
- Add before		✓
- Add after		✓
- Replace		✓
Template Hook Point		✓

Table 11.4: An overview of all used customization approaches.

11.3 Examination Regulation System

A further example is a standalone data-centric application prototype of an examination regulation system for universities, which manages lecture modules and their credits for a particular degree course. In this case example, the focus has been set to develop a functional prototype only. It targets evaluation of the required data structure but adds additional functionality such as an importer for the existing system, notification of responsible editors when a lecture module changes, particular views for students and employees, PDF export, and overviews with multiple charts.

In multiple iterations, as proposed for prototyping (cf. Section 3.3.1), the data structure shown in Figure 11.9 has been identified. Note that this is only an excerpt due to presentational reasons. The full CD4A model is shown in Listing D.4. It consists of an `ExaminationRegulation` that represents the regulations for a particular degree course. Each can exist in multiple versions and, hence, is aware of its predecessors and successors. Changes leading to a new version have to be accepted by a responsible `Employee` managing the degree course. Moreover, each `ExaminationRegulation` consists of multiple `ModuleAreas`, each of which represent the different areas a `Module` is part of, e.g., computer science or mechanical engineering. Such a module can exist in multiple versions as well.

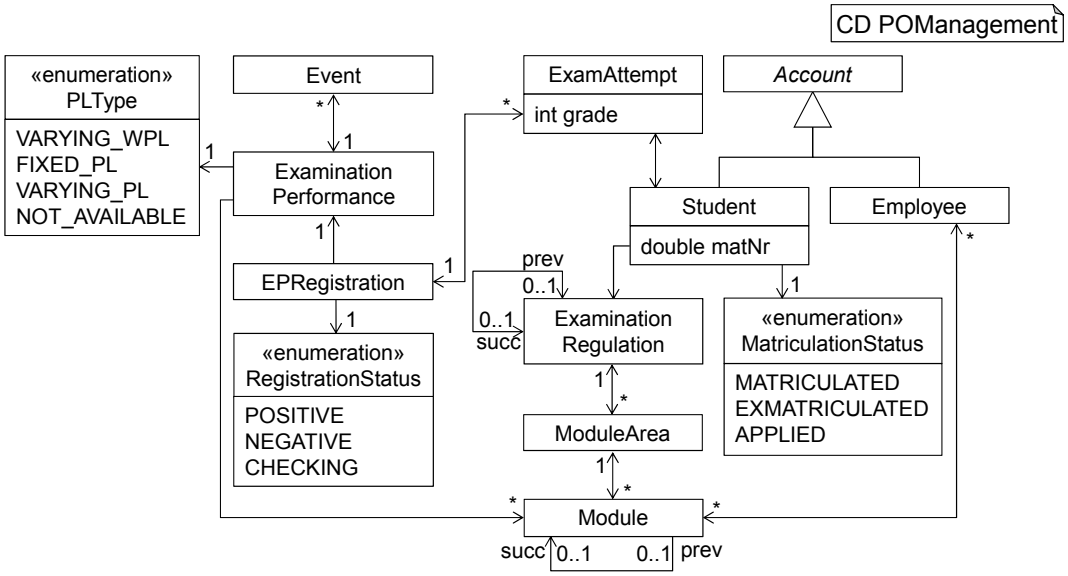


Figure 11.9: The UML CD model for the examination regulation system.

Besides modeling the examination regulation, `Students` are modeled as well to manage their examination regulations and to allow notifications. Each `Student` has a

particular MatriculationStatus, which is required to assign only matriculated students to ExaminationRegulations. Moreover, Students are allowed to take an ExamAttempt 4 times at max. Note that this constraint is not modeled in the data structure but added via handcoded extensions.

Each ExamAttempt has exactly one EPRegistration, which defines the status of the Student's registration for a particular exam. To receive the credits, the exam is associated with a ExaminationPerformance, which defines a particular exam type (PLType) and the Event, when the exam takes place.

11.3.1 Technical Realization

The architecture of the developed prototype consists of all three MontiDEx product layers with additional handcoded extensions, as shown in Figure 11.10. The manually-written code comprises an extension for the ListView (CustomListView component) to provide direct feedback of examination regulations as well as an overview of different reports (StatisticsView component). In addition, the Notification component enables email notifications whenever an examination regulation has been changed. In order to evaluate the prototype with real data, the system provides an importer for the current examination regulation system (ERImporter component) and a PDF exporter to export examination regulations (PDFExpoerter component).

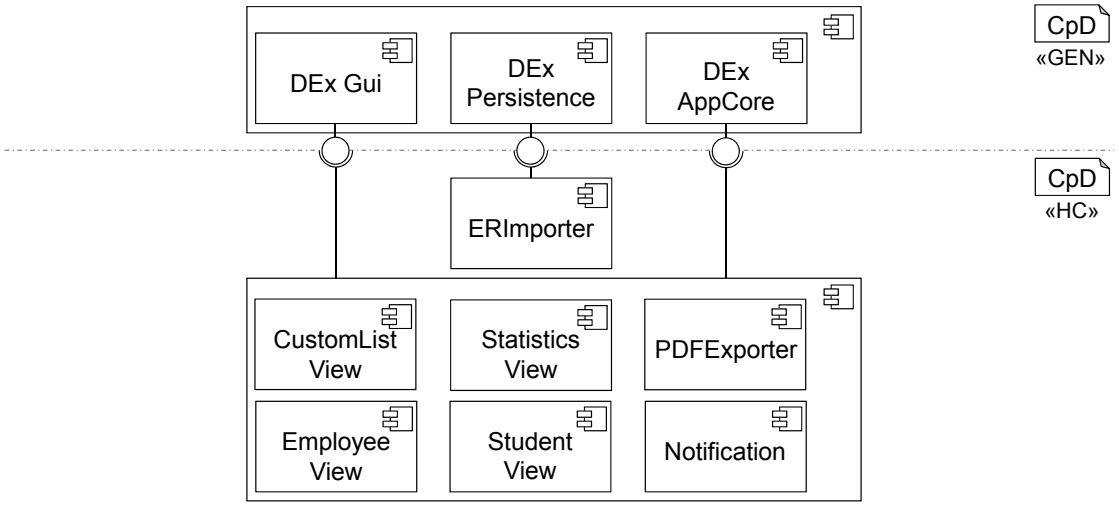


Figure 11.10: Overview of the architecture of the examination regulation system.

The technical realization of this case example was realized within four months by four computer science students in the course of a software lab at the RWTH Aachen university. All students had some experience in Java programming and were undergraduates.

The developed prototype provides distinct views for students and employees (`StudentView` and `EmployeeView` component in Figure 11.10), each of which contains shortcuts for specific user actions and additional information. An example of a student view is shown in Figure 11.11. It contains the tasks (center), a weather overview (upper right corner), and a calender displaying the current curriculum (lower right corner).

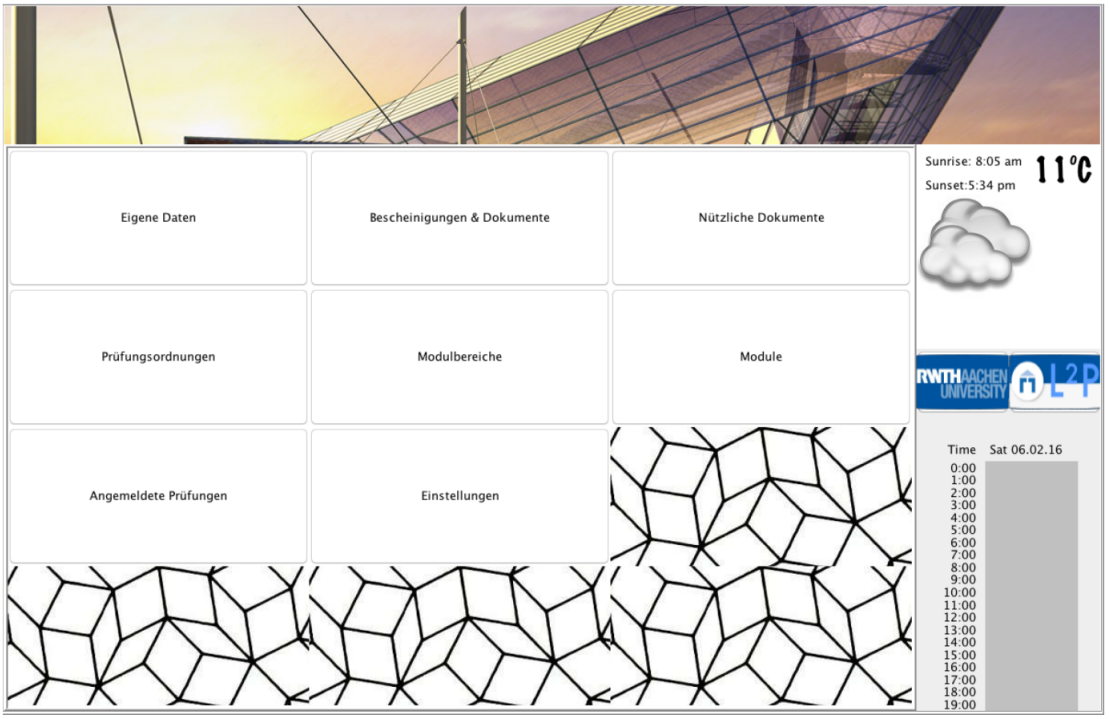


Figure 11.11: A screenshot of the student view extension.

11.3.2 Discussion

In contrast to the case examples in Section 11.1 and Section 11.2, this case example demonstrates the development of a functional prototype using the provided standalone MontiDEx product. Hence, all MontiDEx product layers have been used and extended.

For the developed prototype, this case example shows a potential decrease of LoC to be written as shown in Table 11.5. It shows that the GUI has been manually-written to 18.42 %, which is mainly caused by the added views, reports, and notification support. However, 81.58 % of the required functionality is generated. The persistence layer is generated to 99.71 % and only 0.29 % is manually-written. This is necessary to implement the import and export functionality. Moreover, the application layer is generated to 98.17

% and manually-written to 1.83 % to implement derived associations. As a result, the overall software system is generate to 92.13 % and contains 7.87 % of manually-written LoC. This represents a factor of 11.36 more generated LoC than manually-written.

	LoC		**Percentage**	
	handcoded	generated	handcoded	generated
MontiDEx Product Graphical User Interface Layer	10265	45459	18.42 %	81.58 %
MontiDEx Product Persistence Layer	129	44713	0.29 %	99.71 %
MontiDEx Product Application Layer	662	35429	1.83 %	98.17 %
Sum	**11056**	**125601**	**7.87** %	**92.13** %

Table 11.5: The amount of generated and manually-written LoC for the examination regulation system.

In this case example, the required functionality is added using handcoded extensions provided by the MontiDEx code generator and the MontiDEx generated product, as shown in Table 11.6. In particular, it is sufficient to use the Extended Generation Gap-Pattern, because only the generated product is extended without the need to adapt the MontiDEx code generator. However, the MontiDEx code generator uses the template attachment replace operation to replace the default templates.

Handcoded Extensions Approach	**Used**	**Unused**
Extended Generation Gap		
- Signature Extension	✓	
- Implementation Extension	✓	
- Hot Spots	✓	
Template Attachments		
- Add before		✓
- Add after		✓
- Replace	✓	
Template Extensions		
- Add before		✓
- Add after		✓
- Replace		✓
Template Hook Point		✓

Table 11.6: The customization and adaptation approaches used in the development of the examination regulation system.

Chapter 12

Conclusion

This thesis contributes concepts and methods to a lightweight approach for MDP and MDD of data-centric applications. In this chapter, this thesis is concluded. First, Section 12.1 summarizes this thesis and addresses the raised research question (cf. Section 1.2). Afterwards, we propose potential further research directions in Section 12.2.

12.1 Summary

A data-centric application is a lightweight client application that offers a GUI and access to a persistence infrastructure, and regards management of structured information as the primary concern. The development of data-centric applications has been improved by employing concepts and methods from MDD in various ways (cf. Section 2.4). Hence, this thesis builds on these achievements to propose a lightweight method for data structure prototyping of data-centric applications and a lightweight method for MDD of data-centric applications (cf. Chapter 3). The approach proposed in this thesis differs form existing approaches by explicitly addressing customizability and adaptation concerns, facilitating code generator reuse, supporting underspecification to enable use in early development stages, addressing the needs of different roles involved in the development process (cf. Section 3.1.3), and ensuring data consistency to reduce the gap between description of a data structure and its implementation.

Data-centric applications are generated from a structural description of the managed data. Hence, this thesis addresses modeling concerns of data-centric applications by a language family that comprises the CD4A ML for structural description of managed data, the CD4Code ML for code generation, the ADJava ML for process description. The CD4A ML (cf. Chapter 4) is explicitly designed for description of analysis models, which is an abstract and descriptive model of a problem domain describing structured information. It reduces semantic variation points and language concepts to provide a clear mapping for synthesizing fully executable Java Swing applications. However, due to the foundation of CD4A, i.e., UML/P CD, it still harbors semantic variation points, which are obstructive for code generation. Hence, to achieve effectiveness, semantic

variation points are resolved by defaults, which are predefined choices in the mapping taken by the code generator to resolve underspecification.

The generated data-centric application is realized by a data-centric infrastructure that extends the data structure, which is a generated implementation of a CD4A model, with a GUI, an application core for object management, and access to a persistence infrastructure to manage the created data structure instances. Mappings from CD4A to Java targeting generator developers are proposed in Chapter 5 and in Chapter 7. The data-centric infrastructure uses a three-layered architecture that can be used standalone or in parts due the modular design. The persistence infrastructure to store instantiated objects is generic to support rapid prototyping and development (cf. Section 7.5).

However, to generate executable data-centric applications, defaults and design decisions are chosen that do not always fit and may hamper code generator reuse. Hence, limitations and restrictions set by defaults and design decisions are addressed by customization of the generated source code using the developed Extended Generation Gap-Pattern (cf. Chapter 6). This pattern uses object-oriented inheritance to allow overriding generated functionality and adding new ones. Such customization is supported by hot spots, which mark dedicated spots for adaptation.

This customization is not always practical, e.g., because multiple artifacts have to be extended. Furthermore, it is not always suited when overriding defaults and design decisions, e.g., when overriding private methods. Hence, adaptation and reuse concerns of the code generator is regarded by an integration of transformation- and template-based code generation approach (cf. Chapter 8). It uses an IR, which is described by the CD4Code ML, to represent the object-oriented structure of the generated source code. Exogenous transformations are employed to transform the input model to a CD4Code-AST. Afterwards, endogenous transformations are applied to the CD4Code-AST to produce the targeted object-oriented structure. Because the CD4Code ML is target language independent, target language specific source code is added via template attachments. A template attachment allows to attach a list of templates to an CD4Code-AST node. A default set of templates and the template attachments are executed by the template engine to generate target language source code. Hence, the overall code generation process can be adapted in a black-box way by hook points, which mark dedicated spots in templates designed for adaptation, and by direct manipulation of the template attachments with manually-written templates.

The developed concepts and methods are realized in the MontiDEx code generator (cf. Chapter 9). It uses a modular design to enable realization of code generator product lines for data-centric applications. Furthermore, it provides a script-based configuration, which allows to configure every aspect of language processing and code generation to support code generator reuse. In addition, it supports generator developers in maintaining and extending the generator by providing suitable reporting facilities.

Process automation of the data-centric application is supported by an extended data-centric infrastructure, which also represents a case example for using the developed customization and adaptation mechanisms. It supports process description via the AD-Java ML, which is a ML for UML ADs that embeds Java. ADJava facilitates simplified control flow definition and provides auto-connect capabilities (cf. Chapter 10). ADJava models are executed by interpretation to support process automation after deployment of the data-centric application by modelers and senior application developers. To reduce the implementation effort of process automation during the development of data-centric applications, ADJava models can be enriched with Java source code. A code generator is provided that extracts this added Java source code from ADJava models and generates executable Java source code that is executed by the interpreter. Alternatively, such implementation concerns can be added via handcoded extensions. Because the interpreter uses a hash-based mapping between implementation concerns and the action definition, implementation concerns can be reused. However, this approach has shown several limitations and drawbacks including deployment of the interpreter, necessity of design guidelines for ADJava to avoid non-determinism, and use of string comparisons. Such issues are addressed by methods and guidelines. Furthermore, this case example has shown the applicability of the proposed customization and adaptation approaches, and the benefits of the developed methods to support application developers.

Finally, the developed tools and methods are evaluated in the development of different software systems (cf. Chapter 11). The presented examples demonstrate a framework-like use, prototyping, and agile software development of different data-centric applications. Although the results are not scientifically sound and demand for further evaluation, because they are bound to the chosen implementation, they indicate a reduction of the LoC that would have to be manually-written in the development of data-centric applications. In particular, the number of generated source code LoC of a developed data-centric application ranges from 89.62% to 92.13% of the overall system. Alternatively, this generated source code would have to be manually-written.

12.2 Potential Future Work

The proposed approach forms a foundation for effective development and prototyping of data-centric applications and provides valuable results that show potential research directions in MDD and MDP. Subsequently, we list some identified future work.

Extension of Data-Centric Infrastructure: The introduced approach relies on models and defaults for effective code generation. Hence, adaptations and customizations are added by manually-written code. The case examples have shown that this customization has its limits and may require complete implementation of, e.g., GUIs. Hence, a potential research direction is to further reduce the implementation

effort by extending the data-centric infrastructure. For example, to reduce the need to implement a GUI from scratch, a framework-independent infrastructure for basic GUI functionality can be provided.

Code Generator Product Lines: In [RR15, GMR+16], an approach to implement code generator product lines, which are based on a set of modules, variability regions, and a configuration to create code generator variants has been described. It has been further evaluated to regard explicit interfaces [RRRW15] and use of the ST for transformation composition [MSNRR15a]. Based on these variability modeling approaches and the customization and adaptation mechanisms developed in this thesis, a potential research direction are general code generator product lines. Such general product lines of code generators are only partially addressed by current research. In particular, ongoing research regards business application families [Kul10, KBR12] and formalization of valid extension points [HFMH16].

Synergetic Code Generation: To allow adaptable and flexible code generation, we proposed a synergetic transformation- and template-based code generation approach. However, it has been evaluated for CD4A and Java only. In future work, multiple other input and output language combinations can be evaluated to extend the code generation approach and identify potential drawbacks. For example, by extending the IR with target language concepts to regard target language concerns (e.g., [EBBG12]). However, target language independence is reduced in this case. This research may also address the development of further adaptation approaches for code generators and their current limitation in traceability, as indicated in **??**.

Transformation Libraries: Transformations of the IR are essential elements of the code generation approach used in this thesis. They may be extended or adapted to fit different requirements. In addition, generator developers can contribute different transformations to extend the generated data-centric application. To support reuse, versioning, and collaboration concerns, a potential research direction are transformation libraries (e.g., [BLWPO13]). A first step towards building blocks in code generators has already been proposed (cf. [Bic04]).

Fine-grain Rights and Roles: The role-based access control used in this thesis uses types to assign roles to users. This approach is in-line with data consistency. However, if this approach is extend to more fine-grain permissions, e.g., on association or attribute level, data consistency is at risk. Therefore, a potential research direction is to evaluate and identify role-based access control for such scenarios.

Domain-specific Transformations: The transformations used for code generation are designed using Java. To support development of transformations, current approaches on domain-specific transformation languages [Wei12, HHRW15] may be evaluated for code generation as used in this thesis.

Bibliography

[ABKS13] S. Apel, D. Batory, C. Kästner, and Gunter S. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Publishing Company Inc., 2013. 6.2.2

[ABY14] P. A. Akiki, A. K. Bandara, and Y. Yu. Adaptive Model-Driven User Interface Development Systems. *ACM Comput. Surv.*, 47(1), 2014. 2.4

[ADH+09] N. Aschenbrenner, J. Dreyer, M. Hahn, R. Jubeh, C. Schneider, and A. Zündorf. Building Distributed Web Applications Based on Model Versioning with CoObRa: An Experience Report. In *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*. IEEE Computer Society, 2009. 7.4.2

[AHMM07] D. Akehurst, G. Howells, and K. McDonald-Maier. Implementing associations: UML 2.0 to Java 5. *Software & Systems Modeling*, 6(1), 2007. 5.1, 5.2.11, 7.2.2

[AHW03] W. M. P. Aalst, A. H. M. Hofstede, and M. Weske. Business Process Management: A Survey. In *Business Process Management: International Conference*. Springer Berlin Heidelberg, 2003. 10

[AIM10] L. Atzori, A. Iera, and G. Morabito. The Internet of Things: A Survey. *Comput. Netw.*, 54(15), 2010. 1

[AK03] C. Atkinson and T. Kuhne. Model-driven development: a metamodeling foundation. *IEEE Software*, 20(5), 2003. 2.1

[Aki13] P. A. Akiki. Engineering Adaptive User Interfaces for Enterprise Applications. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. ACM, 2013. 1

[Alf16] H. Alfraihi. Towards Improving Agility in Model-driven Development. In *Joint Proceedings of the Doctoral Symposium and Projects Showcase Held as Part of STAF*, 2016. 2.4

[AT01] J. Ali and J. Tanaka. Implementing the dynamic behavior represented as multiple state diagrams and activity diagrams. *Journal of Computer Science and Information Management*, 2(1), 2001. 10.3

[BCDM14] H. Brunelière, J. Cabot, G. Dupé, and F. Madiot. MoDisco: A model

driven reverse engineering framework. *Information and Software Technology*, 56(8), 2014. 8.2

[BCK12] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 3 edition, 2012. 2.3.1

[BCV05] L. Bettini, S. Capecchi, and B. Venneri. Translating Double Dispatch into Single Dispatch. *Electronic Notes in Theoretical Computer Science*, 138(2), 2005. 2nd Workshop on Object Oriented Developments. 7.1.2

[BCW12] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 1st edition, 2012. 1, 2.1, 2.1, 3.2.1, 4.1

[BDLD11] M. L. Bernardi, G. A. Di Lucca, and D. Distante. A model-driven approach for the fast prototyping of web applications. In *13th IEEE International Symposium on Web Systems Evolution*, 2011. 3.1, 3.3.1

[BDV+16] E. Bousse, T. Degueule, D. Vojtisek, T. Mayerhofer, J. Deantoni, and B. Combemale. Execution Framework of the GEMOC Studio (Tool Demo). In *Proceedings of the Int. Conf. on Soft. Lang. Eng.*, 2016. 10.3.1

[Bet13] L. Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2013. 2.2.4

[Béz05] J. Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2), 2005. 2.1

[BFL13] O. B. Badreddin, A. Forward, and T. C. Lethbridge. Improving Code Generation for Associations: Enforcing Multiplicity Constraints and Ensuring Referential Integrity. In *11th International Conference on Software Engineering Research, Management and Applications*, 2013. 1, 1, 4.2.4, 5, 5.1, 5.2, 5.2.5, 5.2.6, 5.2.6, 5.3, 5.3, 7.2.2

[BGSZ08] M. Bork, L. Geiger, C. Schneider, and A. Zündorf. Towards roundtrip engineering - A template-based reverse engineering approach. In *4th European Conference on Model Driven Architecture - Foundations and Applications*, 2008. 6.1.1

[BHKN96] H. Balzert, F. Hofmann, V. Kruschinski, and C. Niemann. The JANUS Application Development Environment - Generating More than the User Interface. In *Computer-Aided Design of User Interfaces I, Proceedings of the Second International Workshop on Computer-Aided Design of User Interfaces*, 1996. 1, 2.4.2, 7.2.2

[BHS07] F. Buschmann, K. Henney, and D. Schmidt. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*. John Wiley & Sons, 2007. 1.2, 2.3.1

[Bic04] L. Bichler. *Codegeneratoren für MOF-basierte Modellierungssprachen*. PhD thesis, Universität der Bundeswehr München, Germany, 2004. 12.2

[BK10] S. Barat and V. Kulkarni. Developing configurable extensible code generators for model-driven development approach. In *SEKE*. Knowledge Systems Institute Graduate School, 2010. 2.4.2

[Blo08] J. Bloch. *Effective Java*. Prentice Hall PTR, 2 edition, 2008. 5.2.3, 5.2.4

[BLWPO13] F. P. Basso, C. M. Lima Werner, R. M. Pillat, and T. C. Oliveira. How do You Execute Reuse Tasks Among Tools? In *25th International Conference on Software Engineering and Knowledge Engineering*. Knowledge Systems Institute Graduate School, 2013. 12.2

[BME+07] G. Booch, R. Maksimchuk, M. Engle, B. Young, J. Conallen, and K. Houston. *Object-oriented Analysis and Design with Applications*. Addison-Wesley Professional, 3 edition, 2007. 7.1.3

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, 1996. 7

[Boc03a] C. Bock. UML 2 Activity and Action Models. *Journal of Object Technology*, 2(4), 2003. 10.2

[Boc03b] C. Bock. UML 2 Activity and Action Models Part 2: Actions. *Journal of Object Technology*, 2(5), 2003. 10.2

[Boc03c] C. Bock. UML 2 Activity and Action Models Part 3: Control Nodes. *Journal of Object Technology*, 2(6), 2003. 10.2

[Boc04a] C. Bock. UML 2 Activity and Action Models Part 4: Object Nodes. *Journal of Object Technology*, 3(1):27–41, 2004. 10.2

[Boc04b] C. Bock. UML 2 Activity and Action Models Part 5: Partitions. *Journal of Object Technology*, 3(7), 2004. 10.2

[Boc05] C. Bock. UML 2 Activity and Action Models Part 6: Structured Activities. *Journal of Object Technology*, 4(4), 2005. 10.2

[BPdOB13] F. P. Basso, R. M. Pillat, T. C. de Oliveira, and L. B. Becker. Supporting large scale model transformation reuse. ACM, 2013. 8.2.3

[BPdODF14] F. P. Basso, R. M. Pillat, T. C. de Oliveira, and M. D. Del Fabro. Generative adaptation of model transformation assets: experiences, lessons and drawbacks. In *Symposium on Applied Computing*. ACM, 2014. 8.2.3

[BPKR09] B. Berenbach, D. Paulish, J. Kazmeier, and A. Rudorfer. *Software & Systems Requirements Engineering: In Practice*. McGraw-Hill, Inc., 1 edition, 2009. 3.1.3

[BPRFF15] F. P. Basso, R. M. Pillat, F. Roos-Frantz, and R. Z. Frantz. Combining MDE and Scrum on the Rapid Prototyping of Web Information Systems. *Int. J. Web Eng. Technol.*, 10(3), 2015. 1, 3.1.3, 3.3.1, 3.3.1, 4.1

[BS05] A. K. Bhattacharjee and R. K. Shyamasundar. *Validated Code Generation for Activity Diagrams.* Springer Berlin Heidelberg, 2005. 10.3

[BV06] A. Balogh and D. Varró. Advanced Model Transformation Language Constructs in the VIATRA2 Framework. In *ACM Symposium on Applied Computing.* ACM, 2006. 8.2.4

[BW07] T. Baar and J. Whittle. On the Usage of Concrete Syntax in Model Transformation Rules. In *6th International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics.* Springer-Verlag, 2007. 2.2.4

[CAB+94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-oriented Development: The Fusion Method.* Prentice-Hall, Inc., 1994. 4.1.1

[Cac07] J. Cachopo. *Development of Rich Domain Models with Atomic Actions.* PhD thesis, Tech. Univ. of Lisbon, 2007. 5.1

[Cam14] F. Campagne. *The MPS Language Workbench: Volume I.* The MPS Language Workbench. 2014. 2.2.4

[Cas85] A. F. Case. Computer-aided software engineering (CASE): technology for improving software development productivity. *ACM SIGMIS Database*, 17(1), 1985. 2.1

[CCF+15] B. Combemale, B. H.C. Cheng, R. B. France, J.-M. Jézéquel, and B. Rumpe. *Globalizing Domain-Specific Languages*, volume 9400 of *LNCS, Programming and Software Engineering.* Springer International Publishing, 2015. 1, 2.1, 2, 2.1.1

[CCS13] F. Ciccozzi, A. Cicchetti, and M. Sjödin. Exploiting UML Semantic Variation Points to Generate Explicit Component Interconnections in Complex Systems. In *10th International Conference on Information Technology: New Generations (ITNG).* IEEE, 2013. 4

[CD08] M. L. Crane and J. Dingel. Towards a UML Virtual Machine: Implementing an Interpreter for UML 2 Actions and Activities. In *Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, CASCON '08. ACM, 2008. 10.3.1, 10.3.3, 10.5.2

[CE00] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, 2000. 2.2.4, 3.1.3

[CH03] K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches. In *Workshop on the Generative Techniques in the Context Of Model-Driven Architecture*. Online Proceedings, 2003. 2.2.4

[CH06] K. Czarnecki and S. Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3), 2006. 2.2.4

[CHN12] M. R. Chaudron, W. Heijstek, and A. Nugroho. How Effective is UML Modeling ? *Softw. Syst. Model.*, 11(4), 2012. 2.1

[CML14] M. Chen, S. Mao, and Y. Liu. Big Data: A Survey. *Mobile Networks and Applications*, 19(2), 2014. 1

[CN01] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co. Inc., 2001. 3.2.3, 8.3.1

[CT13] Z. Chared and S. S. Tyszberowicz. Projective Template-Based Code Generation. In *CAiSE'13 Forum at the 25th International Conference on Advanced Information Systems Engineering*, volume 998. CEURS-WS.org, 2013. 2.2.4, 2.2.4

[CY91] P. Coad and E. Yourdon. *Object-oriented Analysis*. Yourdon Press, 2 edition, 1991. 4.1.1

[DD06] Z. Diskin and J. Dingel. Mappings, Maps and Tables: Towards Formal Semantics for Associations in UML2. In *Model Driven Engineering Languages and Systems: 9th International Conference, MoDELS 2006*. Springer Berlin Heidelberg, 2006. 5.2.6

[DED08] Y. Diskin, S. M. Easterbrook, and J. Dingel. Engineering Associations: From Models to Code and Back through Semantics. In *46th International Conference on Objects, Components, Models and Patterns*. Springer Berlin Heidelberg, 2008. 5.2.6

[Dij82] E. W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, 1982. 3.2.4

[DKN+15] G. Dévai, M. Karácsony, B. Németh, R. Kitlei, and T. Kozsik. UML model execution via code generation. In *1st International Workshop on Executable Modeling)*, 2015. 10.3

[DMPT10] I. Dejanović, G. Milosavljević, B. Perišić, and M. Tumbas. A Domain-Specific Language for Defining Static Structure of Database Applications. *Computer Science and Information Systems*, 7(3), 2010. 2.4.2

[Dog08] A. Dogar. *Model driven development for enterprise applications*. PhD thesis, Concordia University, Faculty of Engineering and Computer Sci-

ence, Computer Science and Software Engineering, Canada, 2008. 1

[DREP12] D. Di Ruscio, R. Eramo, and A. Pierantonio. Model Transformations. In *Formal Methods for Model-Driven Engineering*, volume 7320 of *LNCS*. Springer Berlin Heidelberg, 2012. 2.2.4

[EBBG12] O. El Beggar, B. Bousetta, and T. Gadi. Automatic code generation by model transformation from sequence diagram of system's internal behavior. *International Journal of Computer and Information Technology*, 1(02), 2012. 2.2.4, 8.2.2, 12.2

[EKBM+03] A. A. El Kalam, S. Benferhat, A. Miège, R. El Baida, F. Cuppens, C. Saurel, P. Balbiani, Y. Deswarte, and G. Trouessin. Organization Based Access Control. In *4th IEEE International Workshop on Policies for Distributed Systems and Networks*. IEEE Computer Society, 2003. 7.4.2

[EKW92] D. W. Embley, B. D. Kurtz, and S. N. Woodfield. *Object-oriented Systems Analysis: A Model-driven Approach*. Yourdon Press, 1992. 4.1.1

[Eli94] A. Eliens. *Principles of Object-Oriented Software Development*. Addison-Wesley Longman Publishing Co. Inc., 1994. 6.2.1, 8.1

[ES07] J. Euzenat and P. Shvaiko. *Ontology Matching*. Springer-Verlag, 2007. 7.6

[EvdSV+13] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. H. Wachsmuth, and J. van der Woning. The State of the Art in Language Workbenches. In *Software Language Engineering*, volume 8225 of *LNCS*. Springer International Publishing, 2013. 2.2

[EW01a] R. Eshuis and R. Wieringa. *A Real-Time Execution Semantics for UML Activity Diagrams*. Springer Berlin Heidelberg, 2001. 10.3

[EW01b] R. Eshuis and R. Wieringa. An Execution Algorithm for UML Activity Graphs. In *4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*. Springer-Verlag, 2001. 10.3

[FBHK+07] J. C. Flores Beltran, B. Holzer, T. Kamann, M. Kloss, S. A. Mork, B. Niehues, K. Thoms, G. Pietrek, and J. Trompeter. *Modellgetriebene Softwareentwicklung. MDA und MDSD in der Praxis*. Entwickler-Press, 2007. 6.1.1

[FBL10] A. Forward, O. Badreddin, and T. Lethbridge. Umple: Towards Com-

bining Model Driven with Prototype Driven Software Development. In *International Symposium on Rapid System Prototyping*, pages 1–7, 2010. 2.4.1

[FBLS12] A. Forward, O. Badreddin, T. C. Lethbridge, and J. Solano. Model-driven Rapid Prototyping with Umple. *Softw. Pract. Exper.*, 42(7), 2012. 1, 2.4.1, 3.1, 3.2.1

[FBY08] J. Fu, F. B. Bastani, and I-L. Yen. *14th Monterey Workshop on Innovations for Requirement Analysis. From Stakeholders' Needs to Formal Designs*, chapter Model-Driven Prototyping Based Requirements Elicitation. Springer Berlin Heidelberg, 2008. 3.1

[FHR08] F. Fieber, M. Huhn, and B. Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5), 2008. 2.1

[Fie00] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. 1

[FKC07] D. F. Ferraiolo, D. R. Kuhn, and R. Chandramouli. *Role-Based Access Control*. Artech House Inc., 2nd edition, 2007. 3.2.1, 7.4.2

[Fow97] M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley series in object-oriented software engineering. Addison Wesley, 1997. 4.1.1, 5.2.6

[Fow03a] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003. 7.3.2

[Fow03b] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition, 2003. 4.1

[Fow10] M. Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010. 1.2, 6.2

[FR07] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *Future of Software Engineering 2007*, 2007. 2.1

[FRS13] R. France, B. Rumpe, and M Schindler. Why it is so hard to use models in software development: observations. *Software & Systems Modeling*, 12(4), 2013. 11

[GAL15] M. Garzón, H. I. Aljamaan, and T. C. Lethbridge. Umple: A framework for model driven development of object-oriented systems. In *22nd*

International Conference on Software Analysis, Evolution, and Reengineering, 2015. 1, 2.4.1

[GDCL03] G. Génova, C. R. Del Castillo, and J. Llorens. Mapping uml associations into java code. *Journal of Object Technology*, 2(5):135–162, 2003. 5.1, 5.2.6, 5.2.6, 5.2.7

[Gén01] G. Génova. Semantics of navigability in UML associations. Technical Report UC3M-TR-CS-2001-06, Computer Science Department, Carlos III University of Madrid, 2001. 4.2.4

[Ges08] D. Gessenharter. Mapping the UML2 Semantics of Associations to a Java Code Generation Model. In *11th International Conference on Model Driven Engineering Languages and Systems*, MoDELS '08. Springer-Verlag, 2008. 5.1, 5.2.11, 5.2.11, 7.2.2

[Ges09] D. Gessenharter. Implementing uml associations in java: A slim code pattern for a complex modeling concept. In *Proceedings of the Workshop on Relationships and Associations in Object-Oriented Languages*, RAOOL '09, pages 17–24. ACM, 2009. 5.1, 5.2.6, 5.2.7

[Ges10] D. Gessenharter. UML Activities at Runtime. In *Advances in Conceptual Modeling – Applications and Challenges*, pages 275–284. Springer Berlin Heidelberg, 2010. 3.2.3, 10.3, 10.3.1

[GGLVG08] J. Guerrero García, C. Lemaigre, J. Vanderdonckt, and J. M. González-Calleros. Model-Driven Engineering of Workflow User Interfaces. In *7th International Conference on Computer-Aided Design of User Interfaces*, 2008. 7.3

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995. 3.2.4, 5.3, 6.1, 6.1.1, 6.1.2, 6.4.1, 7.2.1, 7.3.3, 7.5.1, 8.2.1, 11.2.1

[GHK+15a] T. Greifenberg, K. Hölldobler, C. Kolassa, M. Look, P. Mir Seyed Nazari, K. Müller, A. Navarro Pérez, D. Plotnikov, D. Reiss, A. Roth, B. Rumpe, M. Schindler, and A. Wortmann. A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. *CoRR*, abs/1509.04498, 2015. 1.4, 6, 6.1.1, 6.2

[GHK+15b] T. Greifenberg, K. Hölldobler, C. Kolassa, M. Look, P. Mir Seyed Nazari, K. Müller, A. Navarro Pérez, D. Plotnikov, D. Reiss, A. Roth, B. Rumpe, M. Schindler, and A. Wortmann. *Model-Driven Engineering and Software Development: Third International Conference*, chapter Integration of Handwritten and Generated Object-Oriented Code. Springer Interna-

tional Publishing, 2015. 1.4, 6, 6.1.1, 6.2

[Gil16] A. Gilchrist. *Introducing Industry 4.0*, pages 195–215. Apress, 2016. 1

[Gir08] M. Girschick. Integrating Template-Based Code Generation into Graphical Model Transformation. In *Modellierung 2008*, Berlin, 2008. 8.2.4

[GKR+07] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering*, 2007. 2.2

[GKR+08] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, and Völkel S. Monticore: a framework for the development of textual domain specific languages. In *30th International Conference on Software Engineering*, 2008. 2.2

[GLRR13] A. Ganser, H. Lichter, A. Roth, and B. Rumpe. Proactive Quality Guidance for Model Evolution in Model Libraries. In *ME@MoDELS*, volume 1090 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013. 1.4, 6.1.1

[GLRR15] A. Ganser, H. Lichter, A. Roth, and B. Rumpe. Staged model evolution and proactive quality guidance for model libraries. *Software Quality Journal*, 2015. 1.4, 6.1.1

[GMR+16] T. Greifenberg, K. Müller, A. Roth, B. Rumpe, C. Schulze, and A. Wortmann. Modeling Variability in Template-based Code Generators for Product Line Engineering. In *Modellierung 2016*, LNCS P-254. Bonner Köllen Verlag, 2016. 1.4, 12.2

[Gol11] D. Gollmann. *Computer Security*. Wiley, 2011. 7.4.2

[GR11] D. Gessenharter and M. Rauscher. Code Generation for UML 2 Activity Diagrams: Towards a Comprehensive Model-driven Development Approach. In *Proceedings of the 7th European Conference on Modelling Foundations and Applications*. Springer-Verlag, 2011. 10.3

[Grö10] H. Grönniger. *Systemmodell-basierte Definition objektbasierter Modellierungssprachen mit semantischen Variationspunkten*. PhD thesis, RWTH Aachen University, 2010. 3.2.1, 5.1

[GSR05] L. Geiger, C. Schneider, and C. Reckord. Template- and modelbased code generation for MDA-tools. Technical report, 2005. 8.2, 8.2.4

[GVM09] G. Génova, M. C. Valiente, and M. Marrero. On the difference between analysis and design, and why it is relevant for the interpretation of models in Model Driven Engineering. *Journal of Object Technology*, 8(1), 2009. 4.1, 4.1, 4.1

[GW11] M. Goldberg and G. Wiener. *5th International Conference on Evaluation*

of Novel Approaches to Software Engineering, chapter Generating Code for Associations Supporting Operations on Multiple Instances. Springer Berlin Heidelberg, 2011. 5.1

[Hab15] A. Haber. *MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems*. PhD thesis, RWTH Aachen University, Aachen, 2015. 1.1, 2.2, 10.2.6

[Har12] R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012. 5.1

[HBR00] W. Harrison, C. Barton, and M. Raghavachari. Mapping UML Designs to Java. In *15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00. ACM, 2000. 3.1.3, 5.1, 5.2.3, 5.2.5, 5.2.5, 6.1.1, 6.2.2, 7.2.1

[Her13] C. Herrmann. *Integrierte Software Engineering Services zu effizienten Unterstützung von Entwicklungsprojekten*. PhD thesis, RWTH Aachen University, Aachen, 2013. 1.1, 3.1.3

[HFMH16] N. Harrand, F. Fleurey, B. Morin, and K. E. Husa. ThingML: A Language and Code Generation Framework for Heterogeneous Targets. In *19th International Conference on Model Driven Engineering Languages and Systems*, MODELS '16. ACM, 2016. 12.2

[HHRW15] L. Hermerschmidt, K. Hölldobler, B. Rumpe, and A. Wortmann. Generating Domain-Specific Transformation Languages for Component & Connector Architecture Descriptions. In *2nd International Workshop on Model-Driven Engineering for Component-Based Software Systems*, page 18, 2015. 12.2

[HKA10] F. Heidenreich, J. Kopcsek, and U. Aßmann. *Safe Composition of Transformations*. Springer Berlin Heidelberg, 2010. 8.2.3

[HKGV10] Z. Hemel, L. C. L. Kats, D. M. Groenewegen, and E. Visser. Code generation by model transformation: a case study in transformation modularity. *Software & Systems Modeling*, 9(3), 2010. 2.2.4, 2.2.4, 2.2.4, 2.2.4, 2.2.4, 8, 8.2.2, 8.2.3, 8.4.1

[HL01] H. F. Hofmann and F. Lehner. Requirements engineering as a success factor in software projects. *IEEE Software*, 18(4), 2001. 1

[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. 2.2.1

[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. 2.2.1

[HMZ11] H. Hussmann, G. Meixner, and D. Zuehlke. *Model-Driven Development of Advanced User Interfaces*. Studies in Computational Intelligence. Springer Berlin Heidelberg, 2011. 7.3

[HR00] D. Harel and B. Rumpe. Modeling Languages: Syntax, Semantics and All That Stuff (Part I: The Basic Stuff). Technical Report MCS00-16, Mathematics & Computer Sience, Weizmann Institute Of Sience, 2000. 2.2, 3.2.1

[HR04] D. Harel and B. Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *Computer*, 37(10), 2004. 2.2

[HRR12] A. Haber, J. O. Ringert, and B. Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, 2012. 2.1

[HRW15] K. Hölldobler, B. Rumpe, and I. Weisemöller. Systematically deriving domain-specific transformation languages. In *Conference on Model Driven Engineering Languages and Systems*. ACM/IEEE, 2015. 2.2.4

[HS93] G. M. Høydalsvik and G. Sindre. On the Purpose of Object-oriented Analysis. *SIGPLAN Not.*, 28(10), 1993. 4.1, 4.1

[HVEK07] A. Haase, M. Völter, S. Efftinge, and B. Kolb. Introduction to openarchitectureware 4.1. 2. In *MDD Tool Implementers Forum at TOOLS Europe*. `http://www.dsmforum.org/events/mdd-tif07/oAW.pdf`, 2007. 8.2

[HW03] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co. Inc., 2003. 7.4.3

[Jac04] I. Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley Longman Publishing Co. Inc., 2004. 4.1.1

[JBW+14] E. Jakumeit, S. Buchwald, D. Wagelaar, L. Dan, Á Hegedüs, M. Hermannsdörfer, T. Horn, E. Kalnina, C. Krause, K. Lano, M. Lepper, A. Rensink, L. Rose, S. Wätzoldt, and S. Mazanek. A survey and comparison of transformation tools based on the transformation tool contest. *Science of Computer Programming*, 85, Part A, 2014. 2.2.4

[JC15] M. Jančár and S. Chodarev. A Generative Framework for Development of CRUD-based Linux Desktop Applications. In *13th International Scientific Conference on Informatics*, 2015. 1, 2.4.2

[JHH16] R. Jung, R. Heinrich, and W. Hasselbring. *GECO: A Generator Composition Approach for Aspect-Oriented DSLs*. Springer International Publishing, 2016. 8.1

[Jör13] S. Jörges. *Construction and Evolution of Code Generators: A Model-Driven and Service-Oriented Approach*. LNCS sublibrary: Programming and software engineering. Springer Berlin Heidelberg, 2013. 2.2.4, 2.2.4, 9.4

[JSK12] X. Jin, R. Sandhu, and R. Krishnan. *6th International Conference on Mathematical Methods, Models and Architectures for Computer Network Security*, chapter RABAC: Role-Centric Attribute-Based Access Control. Springer Berlin Heidelberg, 2012. 7.4.2

[Kai99] H. Kaindl. Difficulties in the Transition from OO Analysis to Design. *IEEE Software*, 16(5), 1999. 4.1

[Kaj12] E. Kajan. *Information Technology Encyclopedia and Acronyms*. Springer Berlin Heidelberg, 2012. 1

[Kar08] A. Karagkasidis. Developing GUI Applications: Architectural Patterns Revisited. In *13th Annual European Conference on Pattern Languages of Programming*, 2008. 7.3.2

[KBR11] V. Kulkarni, S. Barat, and U. Ramteerthkar. Early Experience with Agile Methodology in a Model-driven Approach. In *14th International Conference on Model Driven Engineering Languages and Systems*, MODELS'11. Springer-Verlag, 2011. 1, 1.2, 2.1, 6

[KBR12] V. Kulkarni, S. Barat, and S. Roychoudhury. Towards Business Application Product Lines. In *15th International Conference on Model Driven Engineering Languages and Systems*, MODELS'12. Springer-Verlag, 2012. 12.2

[KCW10] D. R. Kuhn, E. J. Coyne, and T. R. Weil. Adding Attributes to Role-Based Access Control. *Computer*, 43(6), 2010. 7.4.2

[Ken02] S. Kent. Model Driven Engineering. In *Integrated Formal Methods*, volume 2335 of *LNCS*. Springer Berlin Heidelberg, 2002. 2.1

[KG10] C. Knieke and U. Goltz. An Executable Semantics for UML 2 Activity Diagrams. In *International Workshop on Formalization of Modeling Languages*, FML '10. ACM, 2010. 10.3

[KGBE06] M. Karow, A. Gehlert, J. Becker, and W. Esswein. On the Transition from Computation Independent to Platform Independent Models. In *12th Americas Conference on Infromation Systems*, 2006. 4.1

[KGK+07] D. Koenig, A. Glover, P. King, G. Laforge, and J. Skeet. *Groovy in Action*. Manning Publications Co., 2007. 3.3

[KKP+09] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler, and S. Völkel. Design Guidelines for Domain Specific Languages. In *9th OOPSLA Workshop on Domain-Specific Modeling*, 2009. 1.1, 3.2.2, 4

[Kla06] C. Klare. Konzeption und Realisierung eines Code-Generators mit Tripel Graph Grammatiken. Master's thesis, University of Paderborn, Department of Computer Science, Paderborn, Germany, 2006. 2.2.4

[KLM+16] C. Kolassa, M. Look, K. Müller, A. Roth, D. Reiß, and B. Rumpe. TUnit - Unit Testing For Template-based Code Generators. In *Modellierung 2016*, 2016. 1.4, 8.4.1

[KNNZ99] T. Klein, U. A. Nickel, J. Niere, and A. Zündorf. From UML to Java And Back Again. Technical Report tr-ri-00-216, University of Paderborn, September 1999. 5.2.6

[KR03] V. Kulkarni and S. Reddy. Separation of Concerns in Model-Driven Development. *IEEE Software*, 20(5), 2003. 8

[KR08] V. Kulkarni and S. Reddy. An Abstraction for Reusable MDD Components: Model-based Generation of Model-based Code Generators. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*. ACM, 2008. 1, 2.4.2

[Kra10] H. Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Soft. Eng., Band 1. Shaker Verlag, 2010. 2.2, 2.2.4, 2.2.4, 2.2.4, 2.3, I

[KRV07] H. Krahn, B. Rumpe, and S. Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Proceedings of Models 2007*, pages 286–300, 2007. 2.2, 2.2.1

[KRV08] H. Krahn, B. Rumpe, and S. Völkel. MontiCore: Modular Development of Textual Domain Specific Languages. In *Proceedings of Tools Europe*, 2008. 2.2

[KRV10] H. Krahn, B. Rumpe, and S. Völkel. Monticore: a framework for compositional development of domain specific languages. In *International Journal on Software Tools for Technology Transfer*, volume 12, 2010. 2.2

[KT08] S. Kelly and J.P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley, 2008. 3.1.3, 3.2.1

[Küh06] T. Kühne. Matters of (Meta-) Modeling. *Software & Systems Modeling*, 5(4), 2006. 1

[Kul10] V. Kulkarni. Raising family is a good practice. In *2nd International Workshop on Feature-Oriented Software Development*. ACM, 2010. 12.2

[Kul16] V. Kulkarni. Model Driven Development of Business Applications: A Practitioner's Perspective. In *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 2016. 2.1, 2.4.2, 3.1.3

[KV10] L. C.L. Kats and E. Visser. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. *SIGPLAN Not.*, 2010. 8.2.3

[KVR02] V. Kulkarni, R. Venkatesh, and S. Reddy. *Generating Enterprise Applications from Models*. Springer Berlin Heidelberg, 2002. 1, 2.4.2

[KWB03] A. G. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co. Inc., 2003. 8

[Lan16] K. Lano. *Agile Model-Based Development Using UML-RSDS*. CRC Press LLC, 2016. 1, 2.4.2, 5.2.3, 5.2.5, 5.2.6, 5.2.7, 5.2.11, 7.3

[LBG13] Y. Laurent, R. Bendraou, and M.-P. Gervais. Executing and Debugging UML Models: An fUML Extension. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13. ACM, 2013. 10.3, 10.3.1

[Let14a] T. C. Lethbridge. Teaching Modeling using Umple: Principles for the Development of an Effective Tool. In *27th Conference on Software Engineering Education and Training*, 2014. 1, 2.4.1

[Let14b] T. C. Lethbridge. Umple: An Open-Source Tool for Easy-To-Use Modeling, Analysis, and Code Generation. In *17th International Conference on Model Driven Engineering Languages and Systems*, 2014. 1, 2.4.1

[LEW+02] M. Loy, R. Eckstein, D. Wood, J. Elliott, and B. Cole. *Java Swing*. O'Reilly Media, 2002. 7.3.4

[LL08] X. Li and Z. Liu. Prototyping System Requirements Model. *Electronic Notes in Theoretical Computer Science*, 207, 2008. 1, 2.4.1

[LL13] J. Ludewig and H. Lichter. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. dpunkt.verlag GmbH, 3 edition, 2013. 3.1, 3.1.1, 3.1.3

[LLHL05] X. Li, Z. Liu, J. He, and Q. Long. *Generating a Prototype from a UML Model of System Requirements*. Springer Berlin Heidelberg, 2005. 1, 2.4.1

[LN16] J. Lekane Nimpa. Integration of UML Activity Diagrams @ Runtime into the Data Explorer. Master's thesis, Software Engineering, RWTH Aachen University, 2016. 10, 10.2.6, 10.3.1, 10.3.3, E.3

[LNPR+13] M. Look, A. Navarro Pérez, J. O. Ringert, B. Rumpe, and A. Wortmann. Black-box Integration of Heterogeneous Modeling Languages for Cyber-Physical Systems. In B. Combemale, J. De Antoni, and R. B. France, editors, *Proceedings of the 1st Workshop on the Globalization of Modeling Languages*, volume 1102 of *CEUR Workshop Proceedings*, 2013. 2.2.1

[Loo17] M. Look. *Unterstützung modellgetriebener, agiler Entwicklung mehrbenutzerfähiger, ubiquitärer Enterprise Applikationen durch Generatoren.* PhD thesis, RWTH Aachen University, Aachen, 2017. 1, 1.1, 2.1.1, 2.2, 2.2.1, 2.3, 2.4, 3.2.1, 7.6

[LSHA08] S. Link, T. Schuster, P. Hoyer, and S. Abeck. Focusing Graphical User Interfaces in Model-Driven Software Development. In *1st International Conference on Advances in Computer-Human Interaction*, 2008. 7.3

[LSM+98] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. In *21st National Information Systems Security Conference*, 1998. 7.4.2

[May14] T. Mayerhofer. *Defining executable modeling languages with fUML.* PhD thesis, Technische Universität Wien, Fakultät für Informatik, Institut für Softwaretechnik und Interaktive Systeme, E188, 2014. 10.3.1

[MBR08] T. Memmel, C. Bock, and H. Reiterer. *Joint Working Conferences on Engineering Interactive Systems*, chapter Model-Driven Prototyping for Corporate Software Specification. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. 3.1

[MCF03] S. J. Mellor, A. N. Clark, and T. Futagami. Model-driven development - guest editor's introduction. *Software, IEEE*, 20(5), 2003. 2.1

[MCvG05] T. Mens, K. Czarnecki, and P. van Gorp. A Taxonomy of Model Transformations. In *Language Engineering for Model-Driven Software Development.* Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), 2005. 2.2.4

[Mel04] S. J. Mellor. *MDA Distilled: Principles of Model-driven Architecture.* Addison-Wesley object technology series. Addison-Wesley, 2004. 4.1, 8.2

[MFBC12] P.-A. Muller, F. Fondement, B. Baudry, and B. Combemale. Modeling Modeling Modeling. *Software & Systems Modeling*, 11(3), 2012. 2.1

[MFM+13] G. Milosavljević, M. Filipović, V. Marsenić, D. Pejaković, and I. De-

janović. Kroki: A Mockup-Based Tool for Participatory Development of Business Applications. In *12th International Conference on Intelligent Software Methodologies, Tools and Techniques*, 2013. 1, 2.4.2

[MGS+13] P. Mohagheghi, W. Gilani, A. Stefanescu, M. A. Fernandez, B. Nordmoen, and M. Fritzsche. Where does model-driven engineering help? Experiences from three industrial cases. *Software & Systems Modeling*, 12(3), 2013. 2.1

[Mil07] D. Milicev. On the Semantics of Associations and Association Ends in UML. *IEEE Trans. Software Eng.*, 33(4), 2007. 5.2.6

[MK09] R. Mohan and V. Kulkarni. Model Driven Development of Graphical User Interfaces for Enterprise Business Applications - Experience, Lessons Learnt and a Way Forward. In *Model Driven Engineering Languages and Systems*, volume 5795 of *LNCS*. Springer Berlin Heidelberg, 2009. 1, 2.2.4, 8, 8.1

[MLK12] T. Mayerhofer, P. Langer, and G. Kappel. A Runtime Model for fUML. In *Proceedings of the 7th Workshop on Models@Run.Time*, MRT '12. ACM, 2012. 10.3

[MP03] G. Milosavljević and B. Perišić. Really Rapid Prototyping of Large-Scale Business Information Systems. In *14th IEEE International Workshop on Rapid Systems Prototyping, 2003. Proceedings*, 2003. 1, 2.4.1, 2.4.2, 3.1.3, 3.3.1, 3.3.1

[MP04] G. Milosavljević and B. Perišić. A Method and a Tool for Rapid Prototyping of Large-Scale Business Information Systems. *Computer Science and Information Systems*, 1(2), 2004. 1, 2.4.1

[MRR11] S. Maoz, J. O. Ringert, and B. Rumpe. *14th International Conference on Model Driven Engineering Languages and Systems*, chapter Semantically Configurable Consistency Analysis for Class and Object Diagrams. Springer Berlin Heidelberg, 2011. 3.2.1

[MS92] S. J. Mellor and S. Shlaer. *Object Life Cycles: Modeling the World In States*. Prentice Hall, 1992. 4.1.1

[MSCB15] J. M. Mottu, S. S. Simula, J. Cadavid, and B. Baudry. Discovering model transformation pre-conditions using automatically generated test models. In *26th International Symposium on Software Reliability Engineering*, pages 88–99, 2015. 8.2.3

[MSHL06] G. Monsieur, M. Snoeck, R. Haesen, and W. Lemahieu. Pim to psm transformations for an event driven architecture in an educational tool. In *European Workshop on Milestones, Models and Mappings for Model-*

Driven Architecture, 2006. 1, 2.4.1, 5.2.3, 7.2.1, 7.3

[MSN17] P. Mir Seyed Nayari. *MontiCore: Efficient Development of Composed Modeling Language Essentials*. PhD thesis, RWTH Aachen University, Aachen, 2017. 2.2, 2.2.1, 2.1, 2.2.1, 3, 2.2.3, 4, 2.2.3, 5, I

[MSNRR15a] P. Mir Seyed Nazari, A. Roth, and B. Rumpe. An Extended Symbol Table Infrastructure to Manage the Composition of Output-Specific Generator Information. In *Proceedings of the Workshop on Domain-Specific Modeling*. ACM, 2015. 1.4, 12.2

[MSNRR15b] P. Mir Seyed Nazari, A. Roth, and B. Rumpe. Management of Guided and Unguided Code Generator Customizations by Using a Symbol Table. In *Proceedings of the Workshop on Domain-Specific Modeling*. ACM, 2015. 1.4

[MSNRR15c] 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. 1.4

[Mül17] K. Müller. *Modellbasierte Unterstützung der Software Evolution im industriellen Kontext*. PhD thesis, RWTH Aachen University, Aachen, 2017. (to appear). 1.4

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4), 1989. 10.3.1

[MvG06] T. Mens and P. van Gorp. A Taxonomy of Model Transformation. *Electron. Notes Theor. Comput. Sci.*, 152, 2006. 2.2.4

[MZ04] T. Maier and A. Zündorf. Yet another association implementation. *2nd International Fujaba Days*, 2004. 5.1, 5.2.7

[NPR13] A. Navarro Pérez and B. Rumpe. Modeling Cloud Architectures as Interactive Systems. In *2nd International Workshop on Model-Driven Engineering for High Performance and Cloud computing*, volume 1118. CEUR Workshop Proceedings, 2013. 2.1

[NS16] N. Nahar and K. Sakib. ACDPR: A Recommendation System for the Creational Design Patterns Using Anti-patterns. In *23rd International Conference on Software Analysis, Evolution, and Reengineering*, volume 4, 2016. 7.2.1

[NZ00] J. Niere and A. Zündorf. *Using Fujaba for the Development of Production Control Systems*. Springer Berlin Heidelberg, 2000. 10.3

[OH07] J. Oldevik and Ø. Haugen. Higher-order transformations for product

lines. *Software Product Line Conference, International*, 2007. 8

[Øs07] K. Østerbye. Design of a Class Library for Association Relationships. In *Symposium on Library-Centric Software Design*, LCSD '07. ACM, 2007. 5.2.6

[PADS12] A. Prout, J. M. Atlee, N. A. Day, and P. Shaker. Code generation for a family of executable modelling notations. *Software & Systems Modeling*, 11(2), 2012. 8, 10.3

[Par71] D. L. Parnas. Information Distribution Aspects of Design Methodology. In *IFIP Congress (1)*, 1971. 5.1

[Par72] D. L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Commun. ACM*, 15(12), 1972. 5.1

[PBCN14] J. Porubän, M. Bacíková, S. Chodarev, and M. Nosál. Pragmatic Model-Driven Software Development from the Viewpoint of a Programmer: Teaching Experience. In *Federated Conference on Computer Science and Information Systems*, 2014. 2.4.2

[PBCN15] J. Porubän, M. Bacíková, S. Chodarev, and M. Nosál. Teaching pragmatic model-driven software development. *Comput. Sci. Inf. Syst.*, 12(2), 2015. 1, 2.4.2

[PBvdL05] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag New York, Inc., 2005. 3.2.3, 8.3.1

[PMDM11] B. Perisić, G. Milosavljević, I. Dejanović, and B. Milosavljević. UML Profile for Specifying User Interfaces of Business Applications. *Comput. Sci. Inf. Syst.*, 8(2), 2011. 2.4.1

[Pre95] W. Pree. *Design Patterns for Object-oriented Software Development.* ACM Press/Addison-Wesley Publishing Co., 1995. 3.2.3, 3.2.4, 6.1.1, 6.3, 8.3.1

[Pre00] W. Pree. *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, chapter Hot-spot-driven framework development. Wiley & Sons, 2000. 6.3

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented Modeling and Design.* Prentice-Hall Inc., 1991. 4.1, 4.1.1

[RBP+14] E. Richa, E. Borde, L. Pautet, M. Bordin, and J. F. Ruiz. Towards Testing Model Transformation Chains Using Precondition Construction in Algebraic Graph Transformation. In *Workshop on Analysis of Model Transformations*, 2014. 8.2.3

[RDJK15] Q. M. Rajpoot, C. Damsgaard Jensen, and R. Krishnan. Attributes Enhanced Role-Based Access Control Model. In *12th International Conference on Trust, Privacy and Security in Digital Business*, 2015. 7.4.2

[RdS15] A. Rodrigues da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43, 2015. 2.1

[Rei15] D. Reiß. *Modellgetriebene generative Entwicklung von Web-Informationssystemen*. PhD thesis, RWTH Aachen University, Aachen, 2015. 1.1, 2.2, 2.4, 8.2.2, 10.1, 10.2.4, 10.3

[REM15] S. Roubi, M. Erramdani, and S. Mbarki. A Model-Driven Approach of User Interface for MVP Rich Internet Application. 2(2), 2015. 2.4.2

[RGLR13] A. Roth, A. Ganser, H. Lichter, and B. Rumpe. Staged Evolution with Quality Gates for Model Libraries. In *DChanges*, volume 1008. CEUR-WS.org, 2013. 1.4, 6.1.1

[Ros97] K. T. Roshan. Team-based access control (tmac): A primitive for applying role-based access controls in collaborative environments. In *Proceedings of the Second ACM Workshop on Role-based Access Control*, RBAC '97. ACM, 1997. 7.4.2

[RR15] A. Roth and B. Rumpe. Towards Product Lining Model-Driven Development Code Generators. In *3rd International Conference on Model-Driven Engineering and Software Development*. Springer International Publishing, 2015. 1.4, 2.2.4, 1, 8.1, 12.2

[RRRW15] J. O. Ringert, A. Roth, B. Rumpe, and A. Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics*, 6(1), 2015. 1.4, 12.2

[RRW14] J. O. Ringert, B. Rumpe, and A. Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*, volume 20 of *Aachener Informatik-Berichte, Software Engineering*. Shaker Verlag, 2014. 2.1

[Rum12] B. Rumpe. *Agile Modellierung mit UML*. Springer, 2012. 1, 4.1.1, 4.2.4, 4.2.4, 4.2.4, 4.2.4, 4.2.5, 4.2.5, 5, 5.1, 5.2, 5.2.3, 5.2.5, 5.2.5, 5.2.5, 5.2.6, 5.2.6, 5.2.7, 5.2.11, 5.2.11, 6.2.2, 7.2.1, 7.6, 9.1

[Rum16] B. Rumpe. *Modeling with UML*. Springer, 2016. 1.1, 4, 4.1, 4.1.1, 4.2.4, 4.2.4, 4.2.4, 4.2.4, 4.2.4, 4.2.5

[RW11] B. Rumpe and I. Weisemöller. A Domain Specific Transformation Language. In *Workshop on Models and Evolution*, volume 11, 2011. 2.2.4

[Sar06] S. Sarstedt. *Semantic Foundation and Tool Support for Model-Driven Development with UML 2 Activity Diagrams.* PhD thesis, Universität Ulm, Fakultät für Informatik, Abteilung Programmiermethodik und Compilerbau, 2006. 3.2.3, 10.1, 10.3

[SBM09] S. Sen, B. Baudry, and J.-M. Mottu. *Automatic Model Generation Strategies for Model Transformation Testing.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. 8.2.3

[SBPM09] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework.* Addison-Wesley Professional, 2 edition, 2009. 5.2.3, 5.2.4, 5.2.5, 5.2.6, 5.2.6, 5.2.10, 5.2.11, 9.1.1

[Sch97] H. A. Schmid. Systematic Framework Design by Generalization. *Communication ACM*, 40(10), 1997. 6.3

[Sch12] M. Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P.* Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012. 1.1, 2.2, 2.2.4, 2.2.4, 2.2.4, 2.3, 4, 4.1.1, 4.2.1, 4.2.5, 4.2.5, 4.2.5, 4.3.1, 5.2.1, 7.6, 8.4.1, 10.2.1, I

[Sco04] K. Scott. *Fast Track UML 2.0.* Apresspod Series. Apress, 2004. 4.1.1

[Sel03] B. Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5), 2003. 3.2.1, 6.1.1, 9.3, 9.4, 9.5

[SF07] T. Schattkowsky and A. Forster. On the Pitfalls of UML 2 Activity Modeling. In *Proceedings of the International Workshop on Modeling in Software Engineering.* IEEE Computer Society, 2007. 10.3.1

[SG16] S. Seifermann and H. Groenda. Survey on Textual Notations for the Unified Modeling Language. In *4th International Conference on Model-Driven Engineering and Software Development.* Springer International Publishing, 2016. 4.2

[SK04] M. Störzer and C. Koppen. PCDiff: Attacking the Fragile Pointcut Problem. In *European Interactive Workshop on Aspects in Software*, 2004. 8.3.2

[SLK06] A. L. Santos, A. Lopes, and K. Koskimies. Modularizing Framework Hot Spots Using Aspects. In *XI Jornadas de Ingeniería del Software y Bases de Datos*, 2006. 6.3, 6.3

[Sol10] J. Solano. *Exploring How Model Oriented Programming Can Be Extended to the User Interface Level.* PhD thesis, University of Ottawa, Ottawa, 2010. 1, 2.4.2

[Som10] I. Sommerville. *Software Engineering.* Addison-Wesley, 9 edition, 2010.

1, 3.1, 3.1.1, 3.1.3, 3.3.1, 9.1

[SPHV10] A. Schramm, A. Preußner, M. Heinrich, and L. Vogel. Rapid UI Development for Enterprise Applications: Combining Manual and Model-driven Techniques. In *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems: Part I*. Springer-Verlag, 2010. 2.4.2

[SPJ+05] G. S. Swint, C. Pu, G. Jung, W. Yan, Y. Koh, Q. Wu, C. Consel, A. Sahai, and K. Moriyama. Clearwater: Extensible, Flexible, Modular Code Generation. In *20th IEEE/ACM international Conference on Automated software engineering*. ACM, 2005. 8.2.2

[SQK06] G. Stevens, G. Quaisser, and M. Klann. *End User Development*, chapter Breaking It Up: An Industrial Case Study of Component-Based Tailorable Software Design. Springer Netherlands, 2006. 7.1.1

[SS16] A. Shatnawi and R. Shatnawi. Generating a language-independent graphical user interfaces from UML models. *Int. Arab J. Inf. Technol.*, 13(3), 2016. 3.3.1

[Sta09] G. Starke. *Effektive Software-Architekturen: ein praktischer Leitfaden*. Hanser, 4 edition, 2009. 2.3.1

[Ste02] P. Stevens. On the interpretation of binary associations in the Unified Modelling Language. *Software and Systems Modeling*, 1(1), 2002. 5.1, 5.2.6, 5.2.10

[Ste08] P. Stevens. *International Summer School: Generative and Transformational Techniques in Software Engineering II*, chapter A Landscape of Bidirectional Model Transformations. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. 2.2.4

[Ste13] F. Steimann. Content over Container: Object-oriented Programming with Multiplicities. In *International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013. ACM, 2013. 5.2.6

[Sub15] A. F. Subahi. *A Business User Model-Driven Engineering Method for Developing Information Systems*. PhD thesis, University of Sheffield, Sheffield, UK, 2015. 1, 2.4.2

[SVC06] T. Stahl, M. Voelter, and K. Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006. 1, 1.2, 2.1, 3.2.1, 3.3.1, 6, 6.1, 6.1.1, (i), 6.1.1, 6.2, 7.1.3, 8.2.4

[TLS13] B. Tang, Q. Li, and R. Sandhu. A multi-tenant RBAC model for collaborative cloud services. In *11th Annual International Conference on*

Privacy, Security and Trust (PST), 2013. 7.4.2

[TM05] D. Travkin and M. Meyer. Generation of Type Safe Association Implementations. In *3rd International Fujaba Days*, volume tr-ri-05-259 of *Technical Report*. University of Paderborn, 2005. 5.1, 5.2.7, 5.2.9

[TS98] R. K. Thomas and R. S. Sandhu. Task-Based Authorization Controls (TBAC): A Family of Models for Active and Enterprise-Oriented Autorization Management. In *11th International Conference on Database Securty XI: Status and Prospects*. Chapman & Hall, Ltd., 1998. 7.4.2

[TSL15] B. Tang, R. Sandhu, and Q. Li. Multi-tenancy Authorization Models for Collaborative Cloud Services. *Concurr. Comput. : Pract. Exper.*, 27(11), 2015. 7.4.2

[UN09] M. Usman and A. Nadeem. Automatic Generation of Java Code from UML Diagrams using UJECTOR. *International Journal of Software Engineering and Its Applications*, 3(2), 2009. 10.3

[Vö11] S. Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering Band 9. 2011. Shaker Verlag, 2011. 2.2

[Van05] J. Vanderdonckt. *A MDA-Compliant Environment for Developing User Interfaces of Information Systems*. Springer Berlin Heidelberg, 2005. 1

[VBD+13] M. Völter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, and G. Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013. 2.1.1, 2.2.4

[VG07] M. Völter and I. Groher. Handling Variability in Model Transformations and Generators. In *7th Workshop on Domain-Specific Modeling*, 2007. 8

[Vis02] E. Visser. Meta-programming with Concrete Object Syntax. In *Generative Programming and Component Engineering*, volume 2487 of *LNCS*. Springer Berlin Heidelberg, 2002. 2.2.4

[vL00] A. van Lamsweerde. Requirements Engineering in the Year 00: A Research Perspective. In *22nd International Conference on Software Engineering*. ACM, 2000. 1

[Vli98] J. Vlissides. *Pattern Hatching: Design Patterns Applied*. Addison-Wesley, 1998. 1.2, 6.2

[Wac09] G. Wachsmuth. A Formal Way from Text to Code Templates. In *Fundamental Approaches to Software Engineering*, volume 5503 of *LNCS*. Springer Berlin Heidelberg, 2009. 2.2.4

[Wei12] I. Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. PhD thesis, RWTH Aachen University, Aachen, 2012. ISBN 978-3-8440-1191-3, Aachener Informatik-Berichte, Software Engineering Band 12, Shaker Verlag. 2.2.4, 2.2.4, 8.2.2, 12.2

[WN94] K. Walden and J.-M. Nerson. *Seamless Object-oriented Software Architecture: Analysis and Design of Reliable Systems*. Prentice-Hall, 1994. 4.1.1

[Wor16] A. Wortmann. *An Extensible Component & Connector Architecture Description Infrastructure for Multi-Platform Modeling*. PhD thesis, RWTH Aachen University, Aachen, 2016. 2, 2.1, 2.2.1, 3.1.3

[WS07] K. Wang and W. Shen. Runtime Checking of UML Association-Related Constraints. In *Fifth International Workshop on Dynamic Analysis*, 2007. 5.2.6

[www15a] Freemarker template language. `http://www.freemarker.org/`, October 2015. 2.2

[www15b] OMG UML Specification. `http://www.omg.org/spec/UML/2.5/`, October 2015. 1, 8.2.1, 10.2.3, 10.2.4, 10.2.5, 10.3.1

[www15c] openarchitectureware. `https://web.archive.org/web/20140225123932/http://www.openarchitectureware.org/index.php`, October 2015. 8.2

[www16a] Acceleo User Guide. `http://www.acceleo.org/doc/obeo/en/acceleo-2.6-user-guide.pdf`, 2016. 2.2.4

[www16b] AngularJS Framework. `https://angularjs.org/`, July 2016. 11.2.1

[www16c] Apache maven. `https://maven.apache.org/`, December 2016. 2

[www16d] Apache Shiro. `http://shiro.apache.org/`, April 2016. 7.4.2

[www16e] Apache Tika Framework. `https://tika.apache.org/`, July 2016. 1

[www16f] Google Android. https://www.android.com/, May 2016. 11.1

[www16g] Metaborg Spoofax. `http://www.metaborg.org/spoofax/`, April 2016. 2.2.4

[www16h] Omg fuml specification. `http://www.omg.org/spec/FUML/`, December 2016. 10.3.1

[www16i] Oracle glassfish application server. `https://glassfish.java.net/`, February 2016. 7.4

[www16j] PlantUML. `http://plantuml.com/`, July 2016. 10.2.4

[www16k] PostgreSQL. http://www.postgresql.org/, February 2016. 7.4

[www16l] SeaweedFS Scalable Distributed File System. https://github.com/chrislusf/seaweedfs, July 2016. 11.2.1

[www16m] Spark Framework. http://sparkjava.com/, July 2016. 11.1.1

[www16n] Spring Boot Framework. http://projects.spring.io/spring-boot/, July 2016. 11.2.1

[YT05] E. Yuan and J. Tong. Attributed Based Access Control (ABAC) for Web Services. In *International Conference on Web Services*. IEEE Computer Society, 2005. 7.4.2

[ZR11a] S. Zschaler and A. Rashid. Symmetric Language-Aware Aspects for Modular Code Generators. Technical Report TR-11-01, King's College, Department of Informatics, 2011. 8, 8.3.2

[ZR11b] S. Zschaler and A. Rashid. Towards Modular Code Generators Using Symmetric Language-aware Aspects. In *1st International Workshop on Free Composition*. ACM, 2011. 1.2, 8, 8.1, 8.3.2

Appendix A

Index of Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
CD4A	UML Class Diagram for Analysis
CD4Code	UML Class Diagram for Code
CRUD	Create Read Update Delete
DAO	Data Access Object
DSL	Domain-specific Language
DSML	Domain-specific Modeling Language
EBNF	Extended Backus-Naur Form
EJB	Enterprise Java Beans
EMF	Eclipse Modeling Framework
GPL	General Purpose Programming Language
GPML	General Purpose Modeling Language
GUI	Graphical User Interface
HCI	Human Computer Interface
HTML	Hypertext Markup Language
IR	Intermediate Representation
InfoSys	Information System
J2EE	Java 2 Platform Enterprise Edition
JDK	Java Development Kit
LHS	Left-Hand Side
LoC	Lines of Code
MC	MontiCore
ML	Modeling Language
MBD	Model-Based Development

MDA	Model-Driven Architecture
MDD	Model-Driven Development
MDE	Model-Driven Engineering
MDP	Model-Driven Prototyping
MontiDEx	MontiCore Data Explorer
MVP	Model-View-Presenter
OOA	Object-Oriented Analysis
PIM	Platform Independent Model
POJO	Plain-Old-Java-Object
PSM	Platform Specific Model
RTE	Run-time Environment
RTF	Rich Text Format
REST	Representational State Transfer
RHS	Right-Hand Side
SC	UML Statechart
SD	UML Sequence Diagram
ST	Symbol Table
UI	User Interface
UML	Unified Modeling Language
UML AD	UML Activity Diagram
UML CD	UML Class Diagram
UML OD	UML Object Diagram
UML OCL	UML Object Constraint Language
UML/P CD	UML/P Class Diagram
UML/P OCL	UML/P Object Constraint Language
UML/P OD	UML/P Object Diagram
YAML	Yet Another Markup Language

Appendix B

Diagram and Listing Tags

Tag	Description
AD	Activity Diagram
ADJava	Activity Diagram with Java
AST-CD	Abstract Syntax Tree Class Diagram
AST-OD	Abstract Syntax Tree Object Diagram
CD	Class Diagram
CD4A	Class Diagram for Analysis Diagram
CpD	Component Diagram
CD4Code	Class Diagram for Code
Groovy	Groovy Script
Java	Java Source Code
MCG	MontiCore Grammar
MCL	MontiCore Languages
SC	Statechart Diagram
SD	Sequence Diagram

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

Grammars

In this chapter, the grammars for the developed DSLs are presented. In particular, first the CD4Code grammar is introduced in Section C.1. Note that the CD4Code grammar is the same used for CD4A. However, to realize CD4A additional context conditions are provided, as shown in Section E.1. Afterwards, the AD grammar is introduced in Section C.2 and the ADJava with embedded Java is presented in Section C.3.

C.1 CD4Code Grammar

MCG

```
1 package de.monticore.umlcd4a;
2
3 /* MCG for CD 4 Analysis, Version Feb. 01st, 2016 */
4 grammar CD4Analysis extends de.monticore.types.Types {
5
6   CDCompilationUnit = ("package" package:(Name& || ".")+ ";")?
7     (ImportStatement)* CDDefinition;
8
9   CDDefinition = "classdiagram" Name "{"
10    (cDClasses:CDClass | CDInterface | CDEnum | CDAssociation)*
11    "}";
12
13  CDClass astimplements ASTCDType = Modifier?  "class" Name
14  ( "extends"  superclass:ReferenceType)?
15  ( "implements" interfaces:(ReferenceType || ",")+ )?
16  ( "{" (CDAttribute | CDConstructor | CDMethod )* "}"
17  | ";" );
18
19  CDInterface astimplements ASTCDType = Modifier? "interface" Name
20    ( "extends" interfaces:(ReferenceType || ",")+ )?
21    ( "{" ( CDAttribute | CDMethod )* "}" | ";" );
22
23  CDEnum astimplements ASTCDType = Modifier? "enum" Name
```

```
24      ( "implements" interfaces:(ReferenceType || ",")+ )?
25      ( "{"
26          (CDEnumConstant || ",")* ";" (CDConstructor | CDMethod)*
27        "}"
28      | ";" );
29
30    CDAttribute = Modifier? Type Name ("=" Value)? ";";
31
32    CDEnumConstant = Name ( "("
33      cDEnumParameters:(CDEnumParameter || ",")+ ")" )?;
34
35    CDEnumParameter = Value;
36
37    CDConstructor = Modifier Name "(" (CDParameter || ",")* ")"
38      ("throws" exceptions:(QualifiedName || "," )+)? ";";
39
40    CDMethod = Modifier ReturnType Name "("
41      (CDParameter || ",")* ")"
42      ("throws" exceptions:(QualifiedName || ",")+)? ";";
43
44    CDParameter = Type (Ellipsis:["..."])? Name;
45
46    CDAssociation = Stereotype?
47      (["association"] | ["composition"])
48      ([Derived:"/"])?  Name?
49      leftModifier:Modifier?
50      leftCardinality:Cardinality?
51      leftReferenceName:QualifiedName
52      leftQualifier:CDQualifier?
53      ("(" leftRole:Name ")")?
54      (   leftToRight:["->"]
55        | rightToLeft:["<-"]
56        | bidirectional:["<->"]
57        | unspecified:["--"] )
58      ("(" rightRole:Name ")")?
59      rightQualifier:CDQualifier?
60      rightReferenceName:QualifiedName
61      rightCardinality:Cardinality?
62      rightModifier:Modifier? ";" ;
63
64    Modifier = Stereotype?
65      (["abstract"] | ["final"] | ["static"]
66        | ["private"] | [private:"-"]
67        | ["protected"] | [protected:"#"]
68        | ["public"] | [public:"+"]
69        | ["derived"] | [derived:"/"] )*;
```

```
70
71   Cardinality = many:["[*]"] | one:["[1]"]
72       | oneToMany:["[1..*]"] | optional:["[0..1]"];
73
74   CDQualifier = "[[" Name "]]" | "[" Type "]";
75
76   Stereotype = "<<" values:(StereoValue || ",")+ ">" ">"
77     // It is not possible to define ">>".
78     {((_input.LT(-2).getLine()==_input.LT(-1).getLine())
79     && (_input.LT(-1).getCharPositionInLine() ==
80           _input.LT(-2).getCharPositionInLine()+1))}?;
81
82   StereoValue = Name ("=" value:String)?;
83
84   Value = SignedLiteral;
85 }
```

Listing C.1: The complete grammar of CD4Code, which is used for code generation, including methods, constructors, and modifiers.

C.2 Activity Diagram Language Grammar

```
1 package de.monticore.umlad;                                    MCG
2
3 grammar ActivityGrammar extends de.monticore.types.Types{
4
5   interface LeftEdgeSide;
6   interface RightEdgeSide;
7   interface EdgeElement extends LeftEdgeSide, RightEdgeSide;
8
9   external Body;
10  external BooleanExpression;
11
12  Names = values:Name ("," values:Name)*;
13
14  VariableDeclaration = type:SimpleReferenceType name:Name;
15
16  InputParameters = "(" declarations:VariableDeclaration
17    ("," declarations:VariableDeclaration)* ")";
18
19  OutputParameters = ":" declarations:VariableDeclaration
20    ("," declarations:VariableDeclaration)* ;
21
```

```
22  Guard = "[" ["else"] "]" | "[" BooleanExpression "]" ;
23
24  ADCompilationUnit = PackageStatement? ImportStatement*
25    ActivityDefinition;
26
27  PackageStatement = "package" QualifiedName ";" ;
28
29  ActivityDefinition =   ("[" precondition:BooleanExpression "]")?
30    "activity" name:Name InputParameters? OutputParameters? "{"
31      AutoConnectMode?
32      ( roles:RoleDeclaration
33        | objects:ObjectNodeDeclaration
34        | simpleActions:SimpleActionDeclaration
35        | callActions: CallBehaviorActionDeclaration
36        | decisionNodes: DecisionNodeDeclaration
37        | mergeNodes : MergeNodeDeclaration
38        | forkNodes : ForkNodeDeclaration
39        | joinNodes : JoinNodeDeclaration
40        | edges:ActivityEdgeSequence
41      )*
42    "}"
43    ( "[[" postcondition:BooleanExpression "]]")?;
44
45  AutoConnectMode = "autoconnect" (["type"]| ["pin"]) ";";
46
47  RoleDeclaration = "role" name:Name "{" actionNames:Names "}" ;
48
49  ObjectNodeDeclaration =
50    "data" type:SimpleReferenceType names:Names ";" ;
51
52  ActionSignature =
53    "action" name:Name InputParameters? OutputParameters? ;
54
55  SimpleActionDeclaration =
56    ("[" localPrecondition:BooleanExpression "]")?
57    ActionSignature
58    (actionBody:Body | ";")
59    ("[[" localPostcondition:BooleanExpression "]]" )?;
60
61  CallBehaviorActionDeclaration = ActionSignature "{"
62    "execute" activityName:QualifiedName
63    ("("adInput:Names")")?
64    ("." adOutput:Names "->" actionOutput:Names)? ";"
65  "}";
66
67  DecisionNodeDeclaration = "decision" names:Names ";" ;
```

```
68
69   MergeNodeDeclaration = "merge" names:Names ";" ;
70
71   ForkNodeDeclaration  = "fork" names:Names ";" ;
72
73   JoinNodeDeclaration = "join" names:Names ";" ;
74
75   ActivityEdgeSequence =  sequence:EdgeSequence ";" ;
76
77   EdgeSequence = leftSide:LeftEdgeSide "->"
78     (elements:EdgeElement  "->")*
79     rightSide:RightEdgeSide ;
80
81   AtomicElement implements EdgeElement = Guard?
82     (  ["initial"]
83      | ["final"]
84      | ["flowfinal"]
85      | Names
86      | nodeName:Name "." nodeParameters:Names);
87
88   CompositeElement implements EdgeElement =
89     AtomicElement ("," EdgeElement);
90
91   SequenceElement implements EdgeElement =
92     Guard? "(" EdgeSequence ")" ("," EdgeElement)? ;
93
94   ForkJoinNotation implements LeftEdgeSide, RightEdgeSide =
95     elements:EdgeElement ("||" elements:EdgeElement)+ Guard?;
96
97   EnclosedForkJoinNotation implements EdgeElement =
98     Guard? "(" ForkJoinNotation ")";
99
100  DecisionMergeNotation implements LeftEdgeSide, RightEdgeSide =
101    elements:EdgeElement ("|" elements:EdgeElement )+ Guard?;
102
103  EnclosedDecisionMergeNotation implements EdgeElement =
104    Guard? "(" DecisionMergeNotation ")";
105 }
```

Listing C.2: The full grammar of the activity diagram modeling language to define workflows.

C.3 Activity Diagram Language Grammar with Embedded Java

```
1 package de.monticore.umlad;                                    MCG
2
3 grammar ActivityGrammarWithJava
4          extends de.monticore.umlad.ActivityGrammar,
5                  de.monticore.java.JavaDSL {
6
7   Body = MethodBody;
8   BooleanExpression = Expression;
9 }
```

Listing C.3: The extended grammar of the activity diagram modeling language with embedded Java.

Appendix D

Examples

In this chapter, the CD4A and ADJava models used throughout this thesis are presented. In particular, the lightweight banking system used to introduce the CD4A ML is shown in Section D.1. The ADJava model used to introduce the ADJava language is presented in Section D.5. Finally, the CD4A models for the case examples are shown in Section D.2, Section D.3, and Section D.4.

D.1 CD4A Model for Banking System

CD4A

```
1  package dex;
2
3  import java.util.Date;
4
5  classdiagram BankingSystem {
6
7    class Customer {
8      String firstName;
9      String lastName;
10     Date birthdate;
11     String city;
12     String street;
13     String country;
14   }
15
16   abstract class Account {
17     long number;
18     int balance = 5;
19     int overdraft;
20   }
21
22   class CheckingAccount extends Account {
23     double fixedInterestRate;
```

```
24   }
25
26   class SavingsAccount extends Account {
27     double effectiveInterestRate;
28   }
29
30   interface Employee;
31
32   class Consultant implements Employee {
33     / String personelId;
34   }
35
36   class Transaction {
37     String reference;
38     Optional<Date> executionDate;
39     int value;
40     /boolean completed;
41   }
42
43   class Deposit {
44     int balance;
45   }
46
47   class Share {
48     String name;
49     int value;
50   }
51
52   enum TransactionType {
53     PERIODIC,
54     ONE_TIME;
55   }
56
57   association [1] Account        <-> [[number]] Consultant;
58   association [1] Account (from) <-> (incoming) Transaction;
59   association [1] Account (to)   <-> (outgoing) Transaction;
60   association [*] Account        <-> Customer [1];
61   association [1] Customer       <-> Deposit [0..1];
62   association [1] Deposit        <-> Share [*] <<ordered>>;
63   association type Transaction    -> TransactionType [1];
64   association / [*] Customer     <-> Transaction [*];
65 }
```

Listing D.1: The complete CD4A model for the simplified banking system example in Figure 4.1

D.2 CD4A Model for the POI Management System

CD4A

```
classdiagram POISystem {

  class POI {
    double longitude;
    double latitude;
    Date created;
    Optional<String> description;
    Optional<String> photopath;
    Optional<String> deprecated;
  }

  enum POIType {
    RADAR, TRAFFIC_JAM, COPNSTRUCTION, ACCIDENT, GENERAL;
  }

  enum Rating {
    VERY_BAD, BAD, NEUTRAL, GOOD, VERY_GOOD;
  }

  class POIRating {
    Date closed;
  }

  class User {
    String name;
    Date registered;
    Date lastLogin;
  }

  association [1] User <-> POIRating;
  association POIRating <-> POI [1];
  association POIRating -> Rating [1];
  association / userrating User -> Rating [1];
  association POI -> POIType [1];
}
```

Listing D.2: The CD4A model for the POI Management System.

D.3 CD4A Model for the Audio and Video Streaming

CD4A

```
classdiagram DexflixSystem {
  class Account {
    String loginName;
    boolean blocked;
    boolean upload;
  }

  enum Role {
    ADMIN, USER;
  }

  class Profile {
    String displayName;
    Date birthDate;
    /int age;
  }

  class Rating {
    int ratingValue;
  }

  class Comment {
    String comment;
    Date creation;
  }

  enum Gender {
    MALE, FEMALE;
  }

  abstract class DexFile {
    boolean deleted;
    /double rating;
  }

  class WatchList {
    String name;
  }

  class WatchListElement {
    int order;
  }
```

```
43
44  class History;
45
46  class ViewState {
47    Date startViewing;
48    Date stopViewing;
49    int pausingSec;
50    int views;
51  }
52
53  class SearchTag {
54    String tag;
55  }
56
57  enum FSK {
58    FSK0, FSK6, FSK12, FSK16, FSK18;
59  }
60
61  class DexFileMetaData {
62    String title;
63    String description;
64  }
65
66  enum Visibility {
67    PRIVATE, PUBLIC;
68  }
69
70  class DexFileContainer extends DexFile {
71    /int consumed;
72  }
73
74  class DexFileElement extends DexFile {
75    int consumed;
76    String url;
77    double duration;
78    boolean over;
79  }
80
81  enum MediaType {
82    VIDEO_MP4, VIDEO_WEBM, VIDEO_OGG,
83    AUDIO_MP3, AUDIO_OGG, AUDIO_WAV,
84    IMAGE_JPEG, IMAGE_PNG, IMAGE_GIF;
85  }
86
87  association Account -> Role [1];
88  association [1] Account <-> Profile [0..1];
```

```
89   association [0..1] Profile <-> WatchList [0..1];
90   association [0..1] Profile <-> History [0..1];
91   association Profile -> Gender [1];
92   association [1] Profile <-> Rating;
93   association [0..1] Profile <-> Comment <<ordered>>;
94   association profilePicture [1] Profile <-> DexFile [0..1];
95   association Rating <-> DexFile [1];
96   association [0..1] DexFile <-> Comment <<ordered>>;
97   association [0..1] WatchList <-> WatchListElement;
98   association WatchListElement -> DexFile [1];
99   association [1] History <-> ViewState <<ordered>>;
100  association ViewState -> DexFile [0..1];
101  association SearchTag <-> DexFile;
102  association DexFile <-> (coverPicture) DexFileMetaData [0..1];
103  association metaData DexFile <-> DexFileMetaData [0..1];
104  association DexFileMetaData -> Visibility [1];
105  association DexFileMetaData -> FSK [1];
106  association DexFileElement -> MediaType [1];
107  association [1] DexFileContainer <->
108                  DexFileElement [1..*] <<ordered>>;
109 }
```

Listing D.3: The CD4A model for the audio and video streaming platform

D.4 CD4A Model for the Examination Regulation System

```
1  classdiagram ExaminationRegulationSystem {                    CD4A
2    class ExaminationRegulation {
3      String reference;
4      String title;
5      String shortDescription;
6      String prefix;
7      String postfix;
8      String linkToDokument;
9      String linkToInformation;
10     Date startDate;
11     Date examinationRevision;
12   }
13
14   enum ERStatus {
15     ER_STATUS_ACTIVE,
16     ER_STATUS_RETIRED,
17     ER_STATUS_IN_CHECK;
```

```
18  }
19
20  class Modulearea {
21    int moduleareaID;
22    String title;
23    String shortDescription;
24  }
25
26  class Module {
27    String reference;
28    String title;
29    String shortDescription;
30    String linkToInformation;
31    int durrationInSemesters;
32    int semester;
33    int cycleSemester;
34    String cycleStartSemester;
35    String content;
36    String educationalObjective;
37    String studyAdvisor;
38    String language;
39    String literatur;
40    String grading;
41  }
42
43  class ExaminationPerformance {
44    String title;
45    String shortDescription;
46    String content;
47    String infoLink;
48    int semester;
49    float creditWorkload;
50    float creditBonus;
51    float contactTime;
52    float selfStudyTime;
53    String reference;
54
55    boolean isModularPerformance;
56  }
57
58  enum SpecialRegistrationType {
59    REGISTRATION_WITH_WRITTEN_APPLIKATION;
60  }
61
62  class Account {
63    String login;
```

```
64    String emailAddress;
65  }
66
67  class Student extends Account {
68    int studentID;
69    int alter;
70    String geschlecht;
71    int cp;
72  }
73
74  class StaffMember extends Account {
75    int staffMemberID;
76  }
77
78  enum Verankerung {
79    SEMESTERVARIABLE_WAHLPFLICHTLEISTUNG,
80    SEMESTERFIXIERTE_PFLICHTLEISTUNG,
81    SEMESTERVARIABLE_PFLICHTLEISTUNG,
82    NICHT_VORHANDEN;
83  }
84
85  class ExamAttempt {
86    Date examDate;
87  }
88
89  enum Grade {
90    GRADE_1_0, GRADE_1_3, GRADE_1_7,
91    GRADE_2_0, GRADE_2_3, GRADE_2_7,
92    GRADE_3_0, GRADE_3_3, GRADE_3_7,
93    GRADE_4_0,
94    GRADE_5_0,
95    GRADE_PLUS,
96    GRADE_MINUS;
97  }
98
99  class ERDiff {
100   String fields;
101   String oldValue;
102   String newValue;
103   Date erRevision;
104 }
105
106 enum ERDiffStatus {
107   WAITING_FOR_REVIEW,
108   REVIEWED;
109 }
```

```
110
111 class Event {
112   String description;
113   Date firstStart;
114   int duration;
115   int count;
116   int daysToNext;
117 }
118
119 class EPRegistration {
120   Date registrationDate;
121   Optional<String> registrationText;
122 }
123
124 enum RegistrationStatus {
125   REGISTRATION_POSITIV,
126   REGISTRATION_NEGATIV,
127   REGISTRATION_IN_CHECK;
128 }
129
130 class Notification {
131   Date createdAt;
132   String text;
133   boolean read;
134 }
135
136 class Matriculation {
137   String semester;
138 }
139
140 enum MatriculationStatus {
141   MATRICULATED,EXMATRICULATED,APPLYED;
142 }
143
144 /*Matriculation*/
145 association Matriculation -> MatriculationStatus [1];
146 association [*]Matriculation <-> Student [1];
147 association [*]Matriculation <-> ExaminationRegulation [1];
148
149 /*Notification*/
150 association [*] Notification -> Account [1];
151 association [*] Notification -> ERDiff [*];
152
153 /*EPRegistration*/
154 association EPRegistration -> RegistrationStatus [1];
155 association [*] EPRegistration <-> ExaminationPerformance [1];
```

```
156  association [*] EPRegistration <-> Student [1];
157
158  /*Event*/
159  association /Event -> Date[*];
160  association [*]Event <-> ExaminationPerformance [*];
161
162  /*Module and Modulearea*/
163  association [0..1] Module (prerequisite) <->
164              (follows) Module [*];
165  association [*] Module (manages_Module) <->
166              (managedBy) StaffMember [*];
167  association [0..1] Modulearea (part_of) <->
168              (contains) Module [*];
169  association [0..1] Module (assigned) <->
170              (uses) ExaminationPerformance [*];
171  association [0..1] Module (modulePerformanceOf) <->
172              (modulePerformance)
173             ExaminationPerformance [0..1];
174
175  /*ExaminationRegulation*/
176  association ExaminationRegulation -> ERStatus[1];
177  association [0..1] ExaminationRegulation (predecessor) <->
178              (successor) ExaminationRegulation [0..1];
179  association [0..1] ExaminationRegulation <-> Modulearea [*];
180  association [0..1] ExaminationRegulation <->
181              (changesInRevisions) ERDiff[*];
182
183  /*Person,Account,Staff,...*/
184  association [*] Student (usedBy) <->
185              (studiesWith) ExaminationRegulation [*];
186  association [*] StaffMember (managedBy)<->
187              (managesER) ExaminationRegulation [*];
188  association /ERDiff  -> StaffMember [*];
189  association [*] Student <-> Module [*];
190  association [*] Student <-> Modulearea [*];
191
192  /*ERDiff*/
193  association ERDiff -> ERDiffStatus [1];
194
195  /*ExaminationPerformance*/
196  association ExaminationPerformance -> Verankerung [1];
197  association ExaminationPerformance ->
198              SpecialRegistrationType [0..1];
199
200  /*ExamAttempt*/
201  association [*] ExamAttempt <-> Student [1];
```

```
202   association [*] ExamAttempt <-> ExaminationPerformance [1];
203   association ExamAttempt -> Grade [1];
204 }
```

Listing D.4: The CD4A model for the Examination Regulation System

D.5 Activity Diagram for Transaction Submission

ADJava

```
1  package dex.activities;
2
3  import dex.submission.Transaction;
4  import dex.submission.Customer;
5
6  activity SubmitTransaction(Customer c) {
7
8    role customer {
9      CreateTransaction;
10   }
11
12   role accountant {
13     ValidateCredit,
14     ValidateTransaction,
15     NotifyCustomer,
16     ProcessTransaction,
17     FraudCheck,
18     FinalizeTransaction;
19   }
20
21   merge m1;
22   join j1;
23
24   // actions
25   action CreateTransaction(Customer t) : Transaction o;
26
27   [t != null]
28   action ValidateCredit(Transaction t) : Transaction o;
29   [[o != null]]
30
31   [t != null]
32   action ValidateTransaction(Transaction t) : Transaction o;
33
34   action NotifyCustomer(Transaction t);
35
```

```
36   action ProcessTransaction(Transaction t) : Transaction o;
37
38   action FraudCheck(Transaction t) : Transaction o;
39
40   action FinalizeTransaction(Transaction t);
41
42   // control and object flow
43   c -> CreateTransaction.t;
44   initial -> CreateTransaction;
45   CreateTransaction.o -> ValidateCredit.t;
46
47   ValidateCredit.o ->
48     [o.getFrom().getCustomer().getCredit() > t.getCredit()]
          ValidateTransaction.t
49     | [else] m1;
50
51   m1 -> NotifyCustomer -> final;
52
53   ValidateTransaction.t ->
54     [isValid] (ProcessTransaction.t || FraudCheck.t) -> j1
55     | [else] m1;
56
57   j1 -> [!valid] m1
58         | [else] FinalizeTransaction.t;
59
60   FinalizeTransaction -> final;
61 }
```

Listing D.5: The ADJava model showing the textual version of Figure 10.1

Appendix E

Context Conditions

This chapter lists all context conditions for the CD4A ML in Section E.1, CD4Code ML in Section E.2, and AD ML in Section E.3.

E.1 CD4A Context Conditions

ID	0xC4A01
Name	DiagramNameUpperCase
Severity	Error
Description	First character of the diagram name must be upper-case.
ID	0xC4A04
Name	UniqueTypeNames
Severity	Error
Description	The name is used several times. Classes, interfaces and enumerations may not use the same names.
ID	0xC4A05
Name	TypeNameUpperCase
Severity	Error
Description	The first character of the type name must be upper-case.
ID	0xC4A06
Name	EnumConstantsUnique
Severity	Error
Description	Duplicate enumeration constant names.
ID	0xC4A07
Name	ExtendsNotCyclic
Severity	Error

Description	An inheritance cycle has been introduced by an `extends` or `implements` construct. Inheritance may not be cyclic.
ID	0xC4A08
Name	ClassExtendsOnlyClasses
Severity	Error
Description	A class may only extend classes.
ID	0xC4A09
Name	InterfaceExtendsOnlyInterfaces
Severity	Error
Description	An interface may only extend interfaces.
ID	0xC4A11
Name	AttributeTypeCompatible
Severity	Error
Description	The value assignment for an attribute is not compatible to its type.
ID	0xC4A12
Name	AttributeNameLowerCase
Severity	Error
Description	An attribute must start in lower-case.
ID	0xC4A13
Name	AttributeOverriddenTypeMatch
Severity	Error
Description	A class overrides an attribute of another class with a different type.
ID	0xC4A14
Name	AttributeTypeExists
Severity	Error
Description	The type of an attribute is unknown.
ID	0xC4A15
Name	AttributeUniqueInClassCoco
Severity	Error
Description	Attributes defined multiple times in class.
ID	0xC4A16
Name	AssociationNameLowerCase
Severity	Error
Description	An association name must start in lower-case.

ID	0xC4A17
Name	AssociationRoleNameLowerCase
Severity	Error
Description	A role name of an association must start with a lower-case.
ID	0xC4A18
Name	CompositionCardinalityValid
Severity	Error
Description	A composition has an invalid cardinality that is greater than one.
ID	0xC4A19
Name	AssociationQualifierTypeExists
Severity	Error
Description	Only external data types and types defined within the class diagram may be used.
ID	0xC4A20
Name	AssociationQualifierAttributeExistsInTarget
Severity	Error
Description	If the qualifier is an attribute qualifier, then it has to exist in the referenced type.
ID	0xC4A21
Name	AssociationSourceNotEnum
Severity	Error
Description	An association is invalid, because an association's source may not be an Enumeration.
ID	0xC4A22
Name	AssociationSourceTypeNotExternal
Severity	Error
Description	An association's source may is an external type.
ID	0xC4A24
Name	AssociationOrderedCardinalityGreaterOne
Severity	Error
Description	An association is invalid, because ordered associations are forbidden for a cardinality lower or equal to 1.
ID	0xC4A25
Name	AssociationNameNoConflictWithAttribute

Severity	Error
Description	An association conflicts with an attribute, i.e., they have the same name.
ID	0xC4A10
Name	ClassImplementOnlyInterfaces
Severity	Error
Description	A class may only implement interfaces.
ID	0xC4A26
Name	AssociationNameUnique
Severity	Error
Description	An association is defined multiple times.
ID	0xC4A27
Name	AssociationRoleNameNoConflictWithAttribute
Severity	Error
Description	The role name of an association conflicts with an attribute's name.
ID	0xC4A28
Name	AssociationRoleNameNoConflictWithOtherRoleNames
Severity	Error
Description	A role name of an association conflicts with a role name of another association.
ID	0xC4A29
Name	GenericsNotNested
Severity	Error
Description	Generic types may not be nested.
ID	0xC4A34
Name	TypeNoInitializationOfDerivedAttribute
Severity	Error
Description	Derived attributes may not be initialized.
ID	0xC4A35
Name	AssociationQualifierOnCorrectSide
Severity	Error
Description	The qualifier of an qualified association is at an invalid position regarding the association's direction.
ID	0xC4A36
Name	AssociationSrcAndTargetTypeExistChecker

Severity	Error
Description	Either the source or the target of an association does not exist, i.e., it is not defined.
ID	0xC4A52
Name	AttributeNotAbstractCoCo
Severity	Error
Description	An attribute cannot be abstract.
ID	0xC4A61
Name	ClassModifierOnlyAbstractCoCo
Severity	Error
Description	The class modifier is not abstract but a different one. Only abstract is permitted.
ID	0xC4A62
Name	ClassNoConstructorsCoCo
Severity	Error
Description	A class has a constructor defined. Classes cannot have constructors.
ID	0xC4A63
Name	ClassNoMethodsCoCo
Severity	Error
Description	A class has a method defined. Classes cannot have methods.
ID	0xC4A64
Name	AttributeModifierOnlyDerivedCoCo
Severity	Error
Description	The modifier of an attribute is only derived.
ID	0xC4A65
Name	InterfaceNoModifierCoCo
Severity	Error
Description	Interface may not have modifiers.
ID	0xC4A66
Name	InterfaceNoAttributesCoCo
Severity	Error
Description	Interface may not have attributes.
ID	0xC4A67
Name	InterfaceNoMethodsCoCo

Severity	Error
Description	Interface may not have methods.
ID	0xC4A68
Name	EnumNoModifierCoCo
Severity	Error
Description	Enumerations may not have any modifier.
ID	0xC4A69
Name	EnumNoConstructorsCoCo
Severity	Error
Description	Enumerations may not have any constructor.
ID	0xC4A70
Name	EnumNoMethodsCoCo
Severity	Error
Description	Enumerations may not have any Method.
ID	0xC4A72
Name	AssociationEndModifierRestrictionCoCo
Severity	Error
Description	Associations cannot have any stereotype other than ≪ordered≫.
ID	0xC4A73
Name	EnumerationNameUpperCase
Severity	Error
Description	The name of each enumeration constant has to be written in upper cases.
ID	0xC4A74
Name	NoInitialValueExpression
Severity	Error
Description	An initial value has to be a primitive value or a String value matching the attributes type.
ID	0xC4A80
Name	AssociationQualifierAttributeExistsInTarget
Severity	Error
Description	The referenced qualifier type cannot be resolved.
ID	0xC4A92
Name	NoExternalSuperClassOrInterface

Severity	Error
Description	External data types that are not defined in another CD4A model cannot be used as super classes or implemented.

E.2 CD4Code Context Conditions

Note that these are additional context conditions for the CD4Code language. The full language specification requires the Section E.1 as well.

ID	0xC4A57
Name	AssociationModifier
Severity	Error
Description	The modifier abstract can not be used for associations.
ID	0xC4A51
Name	InterfaceAttributesStatic
Severity	Error
Description	An attribute in an interface must be static.
ID	0xC4A56
Name	InterfaceInvalidModifiers
Severity	Error
Description	Interfaces may only be `public`.
ID	0xC4A52
Name	AttributeNotAbstract
Severity	Error
Description	Attributes may not be abstract.
ID	0xC4A54
Name	ModifierNotMultipleVisibilities
Severity	Error
Description	Only none or one visibility is supported per modifier.
ID	0xC4A53
Name	ClassInvalidModifiers

Severity	Error
Description	Classes may neither be derived nor static.
ID	0xC4A55
Name	EnumInvalidModifiers
Severity	Error
Description	Enums may only be public (or have no modifier).

E.3 Activity Context Conditions

Note that these context conditions are an adapted version from our previous work [LN16].

ID	0xAD0022
Name	ActivityInputNameNotUsedAsOutputNameCoCo
Severity	Error
Description	The name of an activity input parameter should not be used to name an output parameter of the same activity.
ID	0xAD0023
Name	ActivityInputParameterOverwritingCoCo
Severity	Warning
Description	If an Activity Input parameter is on the right side of an edge, then there is a danger of overwriting its value.
ID	0xAD0025
Name	ActivityInputUsedCoCo
Severity	Error
Description	Each activity input parameter should occurs exactly once on the left side of an edge.
ID	0xAD0028
Name	ActivityNameEqualsSimpleFileNameCoCo
Severity	Error
Description	The name of an activity has to be same as the name of the file containing it textual definition.
ID	0xAD0031
Name	ActivityNameStartsWithUpperCaseLetterCoCo

Severity	Warning
Description	The name of an activity starts with a capital letter.
ID	0xAD0033
Name	ActivityOutputParameterReadCoCo
Severity	Warning
Description	If an activity output parameter occurs on the left side of an edge, null values may be read.
ID	0xAD0034
Name	ActivityOutputUsedCoCo
Severity	Error
Description	Each activity output parameter should occur at least once on the right side of an edge.
ID	0xAD0036
Name	DistinctImportsCoCo
Severity	Warning
Description	Import statements should be distinct.
ID	0xAD0037
Name	NoConflictingImportsCoCo
Severity	Error
Description	Conflicting imports are forbidden, e.g. *import de.monticore.types.CustomType;* and *import de.monticore.CustomType;* are conflicting.
ID	0xAD0038
Name	StarImportNotAllowedCoCo
Severity	Warning
Description	Import statements such as *import de.monticore.types.*;* are not recommended.
ID	0xAD0120
Name	DistinctRoleNamesCoCo
Severity	Error
Description	Role names should be distinct.
ID	0xAD0122
Name	RoleActionsAreDefinedCoCo
Severity	Error

Description	The actions belonging to a role should be defined in the same activity.
ID	0xAD0123
Name	RoleNameDistinctFromActivityNameCoCo
Severity	Warning
Description	The activity name should not be used to name a role.
ID	0xAD0124
Name	RoleNameStartsWithUpperCaseLetterCoCo
Severity	Warning
Description	The name of a role or activity partition should start with a capital letter.
ID	0xAD0105
Name	DistinctDataNodeNamesCoCo
Severity	Error
Description	In an activity, the names of object nodes that are neither input nor output parameters should be distinct.
ID	0xAD0106
Name	DataNodeNameDistinctFromInputAndOutputCoCo
Severity	Error
Description	An object node that is neither an activity input nor output should have a name distinct from the activity input and output parameter names.
ID	0xAD0108
Name	DataNodeNameStartsWithLowerCaseLetterCoCo
Severity	Warning
Description	The name of an object node that is neither an input nor an output parameter should start with a lower case letter.
ID	0xAD0109
Name	DataNodeReadOnceCoCo
Severity	Error
Description	An object node that is neither an activity input nor output parameter should occur exactly once on the left side of an edge.
ID	0xAD0111
Name	DataNodeReadWriteCoCo
Severity	Error

Description	When an object node that is neither an activity input nor output parameter occurs on the left side of an edge (the node is read), then is must also occur on the right side of another edge (the node must be set).
ID	0xAD0114
Name	DistinctInputParameterNamesCoCo
Severity	Error
Description	Activity or action input parameter names should be distinct.
ID	0xAD0116
Name	DistinctOutputParameterNamesCoCo
Severity	Error
Description	Activity or action output parameter names should be distinct.
ID	0xAD0119
Name	ParameterNameStartsWithLowerCaseLetter
Severity	Warning
Description	The name of an activity, action input, or output parameter should start with a lower case letter.
ID	0xAD0039
Name	ControlNodeNameDistinctFromActionNameCoCo
Severity	Error
Description	An action name can not be used to name a control node.
ID	0xAD0040
Name	ControlNodeNameDistinctFromActivityNameCoCo
Severity	Warning
Description	The name of a control node should be distinct from that of the corresponding activity.
ID	0xAD0041
Name	ControlNodeNameDistinctFromRoleNameCoCo
Severity	Error
Description	A role name should not be used to name a control node.
ID	0xAD0042
Name	ControlNodeNameStartsWithUpperCaseLetterCoCo
Severity	Error
Description	The name of an explicitly defined control node should start with an upper case letter.

ID	0xAD0043
Name	DecisionNodeAndDataFlowCoCo
Severity	Error
Description	If the incoming edge of a decision node is a control flow, then all outgoing edges shall be control flows as well. But, if it is an object flow, then all outgoing edges shall be object flows as well.
ID	0xAD0047
Name	DecisionNodeAndElseGuardCoCo
Severity	Error
Description	A decision node has at most one outgoing edge with the guard *else*.
ID	0xAD0049
Name	DecisionNodeHasAtLeastTwoOutgoingEdgesCoCo
Severity	Error
Description	A decision node should have at least 2 outgoing edges.
ID	0xAD0051
Name	DecisionNodeHasOneIncomingEdgeCoCo
Severity	Error
Description	Each decision node has exactly one incoming edge.
ID	0xAD0052
Name	DistinctControlNodeNamesCoCo
Severity	Error
Description	Control node names should be distinct.
ID	0xAD0055
Name	FinalNodeHasNoOutgoingEdgesCoCo
Severity	Error
Description	A final (activity final or flow final) node has no outgoing edges.
ID	0xAD0058
Name	ForkNodeAndDataFlowCoCo
Severity	Error
Description	If the incoming edge of a fork node is a control flow, then all outgoing edges shall also be control flows. But, if it is an object flow, then all outgoing edges shall be object flows as well.

ID	0xAD0061
Name	ForkNodeHasAtLeastTwoOutgoingEdgesCoCo
Severity	Error
Description	A fork node should have at least two outgoing edges.
ID	0xAD0063
Name	ForkNodeHasOneIncomingEdgeCoCo
Severity	Error
Description	A fork node has exactly on incoming edge.
ID	0xAD0065
Name	InitialNodeHasNoIncomingEdgesCoCo
Severity	Error
Description	An initial node has no incoming edges.
ID	0xAD0067
Name	InitialNodeOnlyInvolvedInControlFlowsCoCo
Severity	Error
Description	The outgoing edges of an initial node are control flows only.
ID	0xAD0068
Name	JoinNodeIncomingDataFlowsAndPinCoCo
Severity	Error
Description	Two distinct incoming edges of a join node should not specify the same parameter.
ID	0xAD0070
Name	JoinNodeAndDataFlowCoCo
Severity	Error
Description	If an incoming edge of a join node is a data flow, then the outgoing edge shall be a data flow. If all incoming edges are control flows, then the outgoing edge is also a control flow.
ID	0xAD0073
Name	JoinNodeHasAtLeastTwoIncomingEdgesCoCo
Severity	Error
Description	A join node should have at least 2 incoming edges.
ID	0xAD0074
Name	JoinNodeHasDistinctSourcesCoCo
Severity	Error

Description	The sources of the edges leading to a join node should be distinct.
ID	0xAD0076
Name	JoinNodeHasOneOutgoingEdgeCoCo
Severity	Error
Description	A join node has exactly one outgoing edge.
ID	0xAD0078
Name	MergeNodeAndDataFlowCoCo
Severity	Error
Description	If the outgoing edge of a merge node is a control flow, then all incoming edges shall be control flows. But, if it is an object flow, then all incoming edges shall be object flows.
ID	0xAD0081
Name	MergeNodeDataFlowAndTargetNodeCoCo
Severity	Error
Description	When the outgoing edge of a merge node is a data flow, then the target node of that edge is not a fork, decision or join node.
ID	0xAD0083
Name	MergeNodeHasAtLeastTwoIncomingEdgesCoCo
Severity	Error
Description	A merge node should have at least 2 incoming edges.
ID	0xAD0084
Name	MergeNodeHasDistinctSourcesCoCo
Severity	Error
Description	The sources of the edges leading to a merge node should be distinct.
ID	0xAD0086
Name	MergeNodeHasOneOutgoingEdgeCoCo
Severity	Error
Description	A merge node has exactly one outgoing edge.
ID	0xAD0014
Name	ActionDistinctIncomingEdgesAndPinCoCo
Severity	Error
Description	Two distinct incoming data flows of an action node should not specify the same input parameter.
ID	0xAD0015
Name	ActionDistinctOutgoingEdgesAndPinCoCo

Severity	Error
Description	Two distinct outgoing data flows of an action node should not specify the same output parameter.
ID	0xAD0016
Name	ActionInputNameNotUsedAsOutputNameCoCo
Severity	Error
Description	The name of an action input parameter should not be used to name an output parameter of the same action.
ID	0xAD0017
Name	ActionNameDistinctFromActivityNameCoCo
Severity	Error
Description	The name of an action should be different from the name of the corresponding activity.
ID	0xAD0019
Name	ActionNameDistinctFromRoleNameCoCo
Severity	Error
Description	A role name can not be used to name an action.
ID	0xAD0020
Name	ActionNameStartsWithUpperCaseLetterCoCo
Severity	Warning
Description	An action name starts with an upper case letter.
ID	0xAD0021
Name	DistinctActionNamesCoCo
Severity	Error
Description	Action names should be distinct.
ID	0xAD0001
Name	CallBehaviorActionOutputsDefinedCoCo
Severity	Error
Description	Consider the example shown below. This context condition checks whether *t1,t2* are defined as output parameters of the action and whether *r1,r2* are defined as output parameters of the called activity.

```
1 action CallBehaviorAction(Object in1, Object in2)
2        : Object out1, Object out2{
3   execute CalledActivity(in2,in1).r2,r1 -> t1,t2;
4 }
```

ID	0xAD0004
Name	CallBehaviorActionInputOutputParameterNumberCoCo
Severity	Error
Description	This context condition checks the following properties: • A CallBehavior action and the called activity should have the same number of input parameters. • A CallBehavior action and the called activity should have the same number of output parameters • All input parameters of a CallBehavior action should be passed to the called activity. • All output parameters of the called activity should be mapped to the outputs of the calling action.
ID	0xAD0011
Name	CalledActivityExistsOnFileSystemCoCo
Severity	Error
Description	The name of a called activity should reference an existing file on the file system.
ID	0xAD0013
Name	CalledActivityInputsDefinedCoCo
Severity	Error
Description	Inputs passed to a called activity should be defined as action inputs. Consider the following example: `action MyCallAction(Object input1, Object input2){ execute AnotherActivity(input2,input1); }`
ID	0xAD0088
Name	ActionSymbolReferenceInputCoCo
Severity	Error
Description	When an action name occurs on the right side of an edge, the associated parameter names should be defined as action inputs.
ID	0xAD0090
Name	ActionSymbolReferenceOutputCoCo
Severity	Error

Description	When an action name occurs on the left side of an edge, the associated parameter names should be defined as action outputs.
ID	0xAD0091
Name	AtomicElementAndNamesCoCo
Severity	Error
Description	When a list of names occurs on an edge side, then exactly one of the following should hold: • All names reference either object nodes or activity input/output parameters • All names reference either control nodes or actions
ID	0xAD0094
Name	ControlNodeParameterReadWriteCoCo
Severity	Error
Description	A control node parameter that is read should also be set, e.g., if *ControlNode.param* occurs on the left side of an edge, then *ControlNode.param* must also occurs on the right side of another edge.
ID	0xAD0096
Name	EnclosedDecisionMergeNotationAndGuardCoCo
Severity	Error
Description	Only EnclosedDecisionMergeNotation should have a guard.
ID	0xAD0097
Name	EnclosedForkJoinNotationAndGuardCoCo
Severity	Error
Description	Only EnclosedForkJoinNotation should have a guard.
ID	0xAD0098
Name	EnclosedNotationNotInSequenceCoCo
Severity	Error
Description	EnclosedDecisionMergeNotation and EnclosedForkJoinNotation should not occur in the middle of a sequence.
ID	0xAD0100
Name	GuardOnObjectFlowCoCo
Severity	Error
Description	Guards should only label data flows.
ID	0xAD0101
Name	LeftSideHasNoGuardCoCo

Severity	Error
Description	The left side of an EdgeSequence should not have a guard.
ID	0xAD0103
Name	SelfLoopForbiddenCoCo
Severity	Error
Description	Self loops are forbidden, as they lead to deadlocks.
ID	0xAD0104
Name	SequenceElementAndGuardCoCo
Severity	Error
Description	When a SequenceElement specifies a guard, the referenced EdgeSequence should not specify one.
ID	0xAD0115
Name	DistinctNamesCoCo
Severity	Error
Description	A list of names should contain no duplicate.
ID	0xAD0118
Name	DistinctTargetsCoCo
Severity	Error
Description	Outgoing edges of action and control nodes should have distinct targets.

Appendix F

MontiDEx Hot Spots

The MontiDEx product offers a variety of hot spots in the generated source code that are designed for customization of the generated source code. Hence, in this chapter, an overview of the most important hot spots is given.

F.1 Graphical User Interface

Hot Spot `XControllerEIMP` StartUp	
Hot spot class	`XControllerEIMP` for the domain model `X.cd`
Purpose	Add specific code that is executed before the graphical user interface is started.
Methods to adapt	`startUp()`
Available API	`XController`
Mandatory/Optional	Optional

Hot Spot `XControllerEIMP` Tear Down	
Hot spot class	`XControllerEIMP` for the domain model `X.cd`
Purpose	Add specific code that is executed after the graphical user interface has ended.
Methods to adapt	`tearDown()`
Available API	`XController`
Mandatory/Optional	Optional

Hot Spot `XControllerEIMP` Add Menu Item	
Hot spot class	`XControllerEIMP` for the domain model `X.cd`
Purpose	Extend the existing MenuBar with additional MenuItems.
Methods to adapt	`addMenuItem()`
Available API	`XController`
Mandatory/Optional	Optional

Hot Spot `XControllerEIMP` Add ToolBar Item	
Hot spot class	`XControllerEIMP` for the domain model `X.cd`
Purpose	Extend the existing ToolBar with additional ToolItems.
Methods to adapt	`addToolItem()`
Available API	`XController`
Mandatory/Optional	Optional

Hot Spot `XControllerEIMP` ServerAccess	
Hot spot class	`XControllerEIMP` for the domain model `X.cd`
Purpose	Add the responsible class to access the persistence server.
Methods to adapt	`getServerAccess()`
Available API	`XController`
Mandatory/Optional	Mandatory

Hot Spot `XControllerEIMP` Init Persistence-Mode	
Hot spot class	`XControllerEIMP` for the domain model `X.cd`
Purpose	Configure the DataManager to use the Persistence-Mode.
Methods to adapt	`initPersistenceMode()`
Available API	`XController`
Mandatory/Optional	Mandatory

Hot Spot `MainWindowView` Init Persistence-Mode	
Hot spot class	`MainWindowViewEIMP`
Purpose	Define the layout of the main window including the Tree-View and the TabView.
Methods to adapt	`layoutMainWindow()`
Available API	`MainWindowView`
Mandatory/Optional	Optional

Hot Spot `MainWindowView` Admin Panel	
Hot spot class	`MainWindowViewEIMP`
Purpose	Define the window for the Admin Tool
Methods to adapt	`getAdminPanelItem()`
Available API	`MainWindowView`
Mandatory/Optional	Mandatory

Hot Spot `MainWindowMenuBar` Customize MenuBar	
Hot spot class	`MainWindowMenuBarEIMP`
Purpose	Customize the layout of the MenuBar.
Methods to adapt	`createMenus()`
Available API	`MainWindowMenuBar`
Mandatory/Optional	Optional

Hot Spot `MainWindowToolBar` Customize ToolBar	
Hot spot class	`MainWindowToolBarEIMP`
Purpose	Customize the layout of the ToolBar.
Methods to adapt	`createControls()`
Available API	`MainWindowToolBar`
Mandatory/Optional	Optional

Hot Spot `MainWindowToolBar` Extend ToolBar	
Hot spot class	`MainWindowToolBarEIMP`
Purpose	Using the predefined layout, the ToolBar can be extended with additional ToolBar elements.
Methods to adapt	`addControls()`
Available API	`MainWindowToolBar`
Mandatory/Optional	Optional

Hot Spot `MainWindowContent` Adapt the Content of the MainWindow	
Hot spot class	`MainWindowContentEIMP`
Purpose	Define all elements in the MainWindow.
Methods to adapt	`createContent()`
Available API	`MainWindowContent`
Mandatory/Optional	Optional

Hot Spot `MainWindowContent` Adapt TreeView	
Hot spot class	`MainWindowContentEIMP`
Purpose	Customize the TreeView element in the MainWindow.
Methods to adapt	`createMainWindowConcreteMenu()`
Available API	`MainWindowContent`
Mandatory/Optional	Optional

Hot Spot `MainWindowContent` Customize the TabView	
Hot spot class	`MainWindowContentEIMP`
Purpose	Extend and Customize the existing TabView.
Methods to adapt	`createTabbedPane()`
Available API	`MainWindowContent`
Mandatory/Optional	Optional

Hot Spot `XPanelComponentView` Customize Component Layout	
Hot spot class	`XPanelComponentViewEIMP` for the domain class `X`
Purpose	Customize the layout of a component view, which encapsulates the user interface elements for attributes and associations.
Methods to adapt	`createPanel()`
Available API	`XPanelComponentView`
Mandatory/Optional	Optional

Hot Spot `XPanelComponentPresenter` Add DirtyEventListener	
Hot spot class	`XPanelComponentPresenterEIMP` for the domain class `X`
Purpose	PanelComponents are marked as dirty when the user changed a value. This hot spot can be used to add additional Listeners that listen to such events.
Methods to adapt	`addDirtyStateChangeEventListener()`
Available API	`XPanelComponentView`
Mandatory/Optional	Optional

Hot Spot `XPanelComponentPresenter` Load Data	
Hot spot class	`XPanelComponentPresenterEIMP` for the domain class `X`
Purpose	Customize the loading of data and how the data is set in the user interface elements.
Methods to adapt	`doLoadData()`
Available API	`XPanelComponentView`
Mandatory/Optional	Mandatory

Hot Spot `XPanelComponentPresenter` Save Data	
Hot spot class	`XPanelComponentPresenterEIMP` for the domain class `X`
Purpose	Adapt the condition that tells when data is able to be stored.
Methods to adapt	`canSaveData()`
Available API	`XPanelComponentView`
Mandatory/Optional	Mandatory

Hot Spot `XPanelComponentPresenter` Before Save Data	
Hot spot class	`XPanelComponentPresenterEIMP` for the domain class `X`
Purpose	Add addititional functionality before the data is eventually stored.
Methods to adapt	`doBeforeSaveData()`
Available API	`XPanelComponentView`
Mandatory/Optional	Optional

Hot Spot `XPanelComponentPresenter` Do Save Data	
Hot spot class	`XPanelComponentPresenterEIMP` for the domain class `X`
Purpose	Adapt process of storing data.
Methods to adapt	`doSaveData()`
Available API	`XPanelComponentView`
Mandatory/Optional	Mandatory

Hot Spot `XPanelComponentPresenter` Undo	
Hot spot class	`XPanelComponentPresenterEIMP` for the domain class `X`
Purpose	Define what needs to be done for Undo-Functionality.
Methods to adapt	`undo()`
Available API	`XPanelComponentView`
Mandatory/Optional	Optional

Hot Spot `XPanelComponentPresenter` Redo	
Hot spot class	`XPanelComponentPresenterEIMP` for the domain class `X`
Purpose	Define what needs to be done for Redo-Functionality.
Methods to adapt	`redo()`
Available API	`XPanelComponentView`
Mandatory/Optional	Optional

Hot Spot `XPanelComponentPresenter` Delete	
Hot spot class	`XPanelComponentPresenterEIMP` for the domain class `X`
Purpose	Define what needs to be done for Delete-Functionality.
Methods to adapt	`delete()`
Available API	`XPanelComponentView`
Mandatory/Optional	Optional

Hot Spot `XPanelComponentPresenter` Clear	
Hot spot class	`XPanelComponentPresenterEIMP` for the domain class `X`
Purpose	Define what needs to be done for Clear-Functionality.
Methods to adapt	`clear()`
Available API	`XPanelComponentView`
Mandatory/Optional	Optional

Hot Spot `XPanelComponentPresenter` Clear	
Hot spot class	`XPanelComponentPresenterEIMP` for the domain class `X`
Purpose	Define when the input in the View is valid.
Methods to adapt	`validateInput()`
Available API	`XPanelComponentView`
Mandatory/Optional	Optional

Hot Spot `XEditPanelModel` Update Data On Server	
Hot spot class	`XEditPanelModelEIMP` for the domain class `X`
Purpose	Adapt the behavior to update the changes of the model to the server.
Methods to adapt	`update()`
Available API	`XEditPanelModel`
Mandatory/Optional	Optional

Hot Spot `XEditPanelModel` Reload Model	
Hot spot class	`XEditPanelModelEIMP` for the domain class `X`
Purpose	Customize what needs to be done when the model is reloaded from the server.
Methods to adapt	`reloadModel()`
Available API	`XEditPanelModel`
Mandatory/Optional	Optional

Hot Spot `XEditPanelModel` Do Before Store	
Hot spot class	`XEditPanelModelEIMP` for the domain class `X`
Purpose	Add functionality that is executed before the model is stored on the server.
Methods to adapt	`doBeforeStore()`
Available API	`XEditPanelModel`
Mandatory/Optional	Optional

Hot Spot `XEditPanelModel` Store	
Hot spot class	`XEditPanelModelEIMP` for the domain class `X`
Purpose	Adapt the behavior that defines how to store elements on the server.
Methods to adapt	`store()`
Available API	`XEditPanelModel`
Mandatory/Optional	Optional

Hot Spot `XEditPanelView` Customize Layout	
Hot spot class	`XEditPanelViewEIMP` for the domain class `X`
Purpose	Customize the layout and of the edit view.
Methods to adapt	`addPanelComponents()`
Available API	`XEditPanelView`
Mandatory/Optional	Optional

Hot Spot `XEditPanelPresenter` Define the View Instance	
Hot spot class	`XEditPanelPresenterEIMP` for the domain class `X`
Purpose	Define the instance of the EditPanelView that is used in the Presenter.
Methods to adapt	`getView()`
Available API	`AbstractEditPanelPresenter`
Mandatory/Optional	Mandatory

Hot Spot `XEditPanelPresenter` Define the Model Instance	
Hot spot class	`XEditPanelPresenterEIMP` for the domain class `X`
Purpose	Define the instance of the EditPanelModel that is used in the Presenter.
Methods to adapt	`getModel()`
Available API	`AbstractEditPanelPresenter`
Mandatory/Optional	Mandatory

Hot Spot `XEditPanelPresenter` Define Attached Listeners	
Hot spot class	`XEditPanelPresenterEIMP` for the domain class `X`
Purpose	Define what action listeners are used for the different elements in the user interface.
Methods to adapt	`doAttachListeners()`
Available API	`AbstractEditPanelPresenter`
Mandatory/Optional	Optional

Hot Spot `XEditPanelPresenter` Set ToolTips	
Hot spot class	`XEditPanelPresenterEIMP` for the domain class `X`
Purpose	Define the ToolTips for elements in the EditPanelView.
Methods to adapt	`setToolTips()`
Available API	`AbstractEditPanelPresenter`
Mandatory/Optional	Optional

Hot Spot `XEditPanelPresenter` Define After Save Behavior	
Hot spot class	`XEditPanelPresenterEIMP` for the domain class `X`
Purpose	Define the behavior that is executed after the object is stored or updated successfully.
Methods to adapt	`doAfterSaveData()`
Available API	`AbstractEditPanelPresenter`
Mandatory/Optional	Optional

Hot Spot `XEditPanelPresenter` Define Action Before Tab is closed	
Hot spot class	`XEditPanelPresenterEIMP` for the domain class `X`
Purpose	Define the behavior that is executed shortly before the tab is closed.
Methods to adapt	`tabAboutToBeClosed()`
Available API	`AbstractEditPanelPresenter`
Mandatory/Optional	Optional

Hot Spot `XEditPanelPresenter` Define when input is valid	
Hot spot class	`XEditPanelPresenterEIMP` for the domain class `X`
Purpose	Allow to define the validity of the object.
Methods to adapt	`isInputValid()`
Available API	`AbstractEditPanelPresenter`
Mandatory/Optional	Mandatory

Hot Spot `XEditPanelPresenter` Adapt Update Controls	
Hot spot class	`XEditPanelPresenterEIMP` for the domain class `X`
Purpose	Defines how the controls are updated after the user interface has changed.
Methods to adapt	`updateControls()`
Available API	`AbstractEditPanelPresenter`
Mandatory/Optional	Mandatory

Hot Spot `XEditPanelPresenter` Remove Element from TabView	
Hot spot class	`XEditPanelPresenterEIMP` for the domain class `X`
Purpose	All opened Tabs correspond to a particular object. If these are closed they have to be unregistered. This method allows to define how this unregistration is done.
Methods to adapt	`handBackIdent()`
Available API	`AbstractEditPanelPresenter`
Mandatory/Optional	Mandatory

Hot Spot `XEditPanelPresenter` Adapt Component Refresh	
Hot spot class	`XEditPanelPresenterEIMP` for the domain class `X`
Purpose	Components are refresched whenever the user interface is updated. This method specifies how this refresh works.
Methods to adapt	`refreshComponent()`
Available API	`AbstractEditPanelPresenter`
Mandatory/Optional	Optional

Hot Spot `XEditPanelPresenter` Load Data of PanelComponents	
Hot spot class	`XEditPanelPresenterEIMP` for the domain class `X`
Purpose	Specifiy how to load data when an edit view is opened.
Methods to adapt	`doLoadData()`
Available API	`AbstractEditPanelPresenter`
Mandatory/Optional	Mandatory

Hot Spot `XListPanelView` Specify Layout	
Hot spot class	`XListPanelViewEIMP` for the domain class `X`
Purpose	Specify the layout of the user interface.
Methods to adapt	`createPanel()`
Available API	`AbstractListPanelView`
Mandatory/Optional	Optional

Hot Spot `XListPanelView` Define the banner	
Hot spot class	`XListPanelViewEIMP` for the domain class `X`
Purpose	Define the Banner for list views.
Methods to adapt	`createBanner()`
Available API	`AbstractListPanelView`
Mandatory/Optional	Optional

Hot Spot `XListPanelView` Header Layout	
Hot spot class	`XListPanelViewEIMP` for the domain class `X`
Purpose	Layout of the header.
Methods to adapt	`createHeader()`
Available API	`AbstractListPanelView`
Mandatory/Optional	Optional

Hot Spot `XListPanelView` Table Layout	
Hot spot class	`XListPanelViewEIMP` for the domain class `X`
Purpose	Specify the table layout of the list view body.
Methods to adapt	`createTablePanel()`
Available API	`AbstractListPanelView`
Mandatory/Optional	Optional

Hot Spot `XListPanelModel` Override Table Model	
Hot spot class	`XListPanelModelEIMP` for the domain class `X`
Purpose	Override the default table model.
Methods to adapt	`getDomainSpecificTableModel()`
Available API	`AbstractListPanelModel`
Mandatory/Optional	Mandatory

Hot Spot `XListPanelModel` Define Remove	
Hot spot class	`XListPanelModelEIMP` for the domain class `X`
Purpose	Define the action to be done when removing an object.
Methods to adapt	`removeObjectFromManagerAndSynchronize()`
Available API	`AbstractListPanelModel`
Mandatory/Optional	Optional

Hot Spot `XListPanelPresenter` Override Table Model	
Hot spot class	`XListPanelPresenterEIMP` for the domain class `X`
Purpose	Override the default table model.
Methods to adapt	`getDomainSpecificTableModel()`
Available API	`AbstractListPanelModel`
Mandatory/Optional	Mandatory

Hot Spot `XListPanelPresenter` Load Data	
Hot spot class	`XListPanelPresenterEIMP` for the domain class `X`
Purpose	Define how to retrieve the data.
Methods to adapt	`loadData()`
Available API	`AbstractListPanelPresenter`
Mandatory/Optional	Optional

Hot Spot `XListPanelPresenter` Execute Delete Action	
Hot spot class	`XListPanelPresenterEIMP` for the domain class `X`
Purpose	Allows to add an additional delete action to be executed.
Methods to adapt	`deleteActionMethod(ActionEvent e)`
Available API	`AbstractListPanelPresenter`
Mandatory/Optional	Optional

Hot Spot `XListPanelPresenter` Remove Object	
Hot spot class	`XListPanelPresenterEIMP` for the domain class `X`
Purpose	Specify how to remove objects from the list view.
Methods to adapt	`removeObject(Collection<X> objects)`
Available API	`AbstractListPanelPresenter`
Mandatory/Optional	Optional

Hot Spot `XListPanelPresenter` Search Listeners	
Hot spot class	`XListPanelPresenterEIMP` for the domain class `X`
Purpose	Attach listeners to search field.
Methods to adapt	`attachListenerToSearchField()`
Available API	`AbstractListPanelPresenter`
Mandatory/Optional	Optional

Hot Spot `XListPanelPresenter` Listeners for Table	
Hot spot class	`XListPanelPresenterEIMP` for the domain class `X`
Purpose	Attach additional listeners to the JTable.
Methods to adapt	`attachListenersToJTable()`
Available API	`AbstractListPanelPresenter`
Mandatory/Optional	Optional

Hot Spot `XListPanelPresenter` Set ToolTips	
Hot spot class	`XListPanelPresenterEIMP` for the domain class `X`
Purpose	Set the tool tips for the list view elements.
Methods to adapt	`setToolTipTexts()`
Available API	`AbstractListPanelPresenter`
Mandatory/Optional	Optional

Hot Spot `XListPanelPresenter` Define Status update	
Hot spot class	`XListPanelPresenterEIMP` for the domain class `X`
Purpose	Define the message that is printed to the StatusBar after the items have been loaded.
Methods to adapt	`updateStatus()`
Available API	`AbstractListPanelPresenter`
Mandatory/Optional	Optional

Hot Spot `XListPanelPresenter` Define TabTitle	
Hot spot class	`XListPanelPresenterEIMP` for the domain class `X`
Purpose	Define the title of the tab for the list view.
Methods to adapt	`getTabTitle()`
Available API	`AbstractListPanelPresenter`
Mandatory/Optional	Optional

Hot Spot `XListPanelPresenter` Define Action Before Close Tab	
Hot spot class	`XListPanelPresenterEIMP` for the domain class `X`
Purpose	Add additional action before the tab is closed.
Methods to adapt	`tabAboutToBeClosed()`
Available API	`AbstractListPanelPresenter`
Mandatory/Optional	Optional

F.2 Application Core

Hot Spot `XBuilder` Specify Validity	
Hot spot class	`XBuilderEIMP` for the domain class `X`
Purpose	Define when an object is valid so that it can be stored.
Methods to adapt	`isValid()`
Available API	`XBuilder`
Mandatory/Optional	Optional

F.3 Persistence

Hot Spot `XStorageBuilder` Building Proxy	
Hot spot class	`XStorageBuilderEIMP` for the domain class `X`
Purpose	Specify how to build a Proxy for the type X.
Methods to adapt	`buildProxy()`
Available API	`XStorageBuilder`
Mandatory/Optional	Optional

Hot Spot `XStorageBuilder` Convert to Proxy	
Hot spot class	`XStorageBuilderEIMP` for the domain class `X`
Purpose	On receiving an object in generic form, define how to map it to a real proxy object of type X.
Methods to adapt	`proxiedOf()`
Available API	`XStorageBuilder`
Mandatory/Optional	Optional

Hot Spot `XStorageBuilder` Convert to Full Object	
Hot spot class	`XStorageBuilderEIMP` for the domain class `X`
Purpose	On receiving an object in generic form, define how to map it to a real object of type X.
Methods to adapt	`of()`
Available API	`XStorageBuilder`
Mandatory/Optional	Optional

Hot Spot `XProxy` Define Loading	
Hot spot class	`XProxyEIMP` for the domain class `X`
Purpose	Define behavior to load full objects from server.
Methods to adapt	`loadObjectIfNecessary()`
Available API	`XProxy`
Mandatory/Optional	Optional

Hot Spot `XServerAccess` Conext Identifier Definition	
Hot spot class	`XServerAccessEIMP` for the domain model `X.cd`
Purpose	Define which project identifier is used for the server.
Methods to adapt	`getContextIdentifier()`
Available API	`XServerAccess`
Mandatory/Optional	Optional

Appendix G

MontiDEx Package Structure

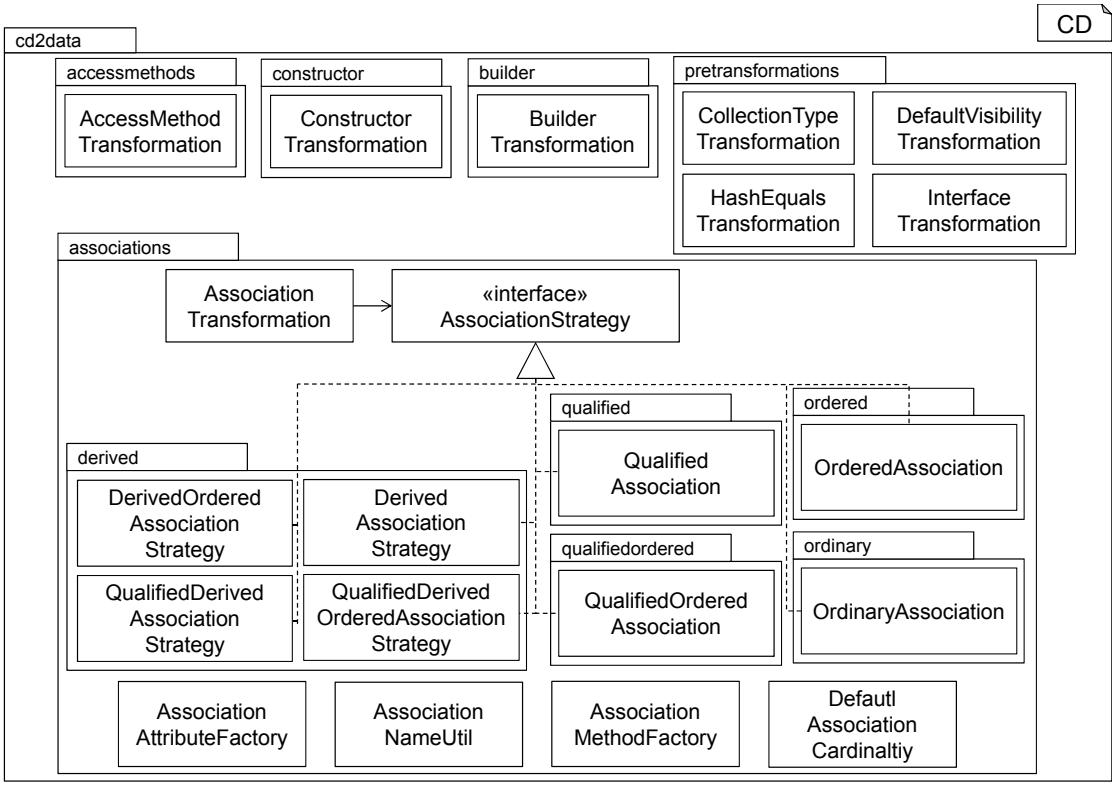


Figure G.1: An overview of the `cd2data` package of the MontiDEx code generator.

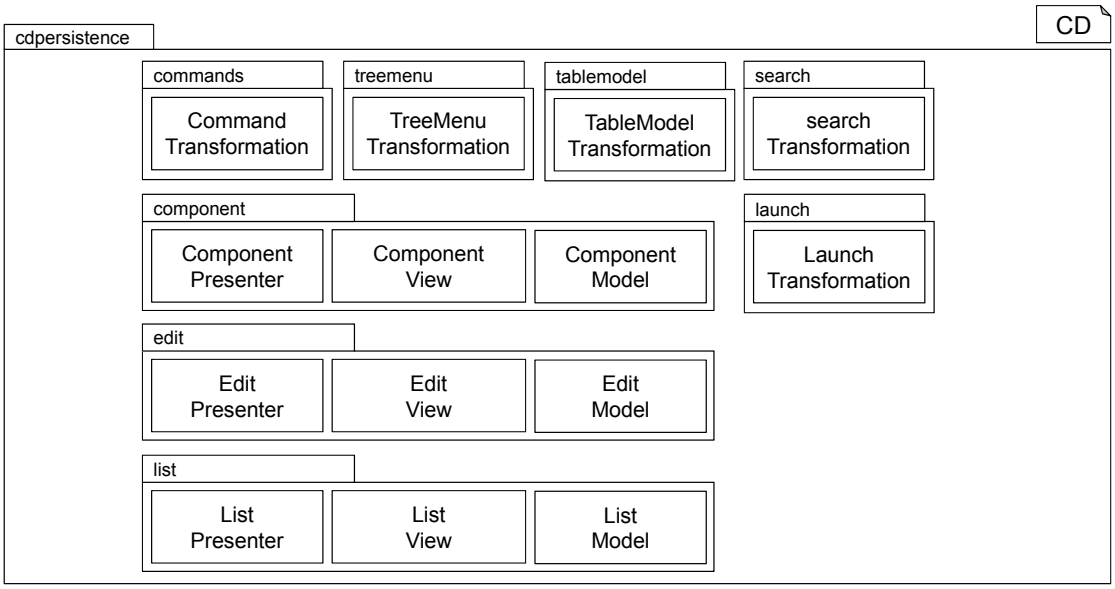


Figure G.2: An overview of the `cd2swing` package of the MontiDEx code generator.

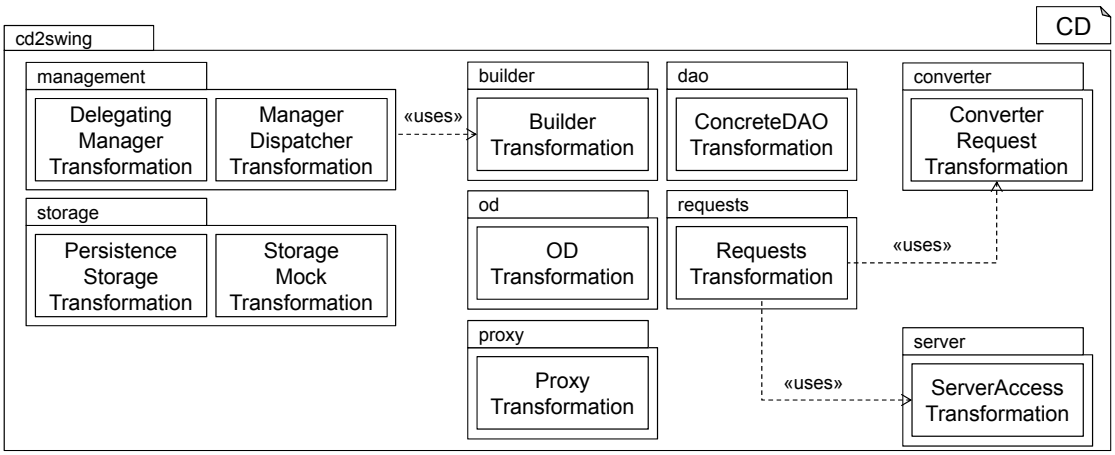


Figure G.3: An overview of the `cd2persistence` package of the MontiDEx code generator.

Appendix H

MontiDEx Hook Points

Hook Point Name	Template	Comment
SetMethodBody:addCheck	SetMethodBody.ftl	It allows to add arbitrary valid Java source code to the mutator methods.
ClassAttribute:addAnnotations	Attribute.ftl	Enables to define annotations for attributes.
ClassContent:addComment	Class.ftl	Add comment to a generated Java class.
ClassContent:addImports	Class.ftl	Add additional imports to a Java class.
ClassContent:addAnnotations	Class.ftl	Add annotations to a Java class.
ClassContent:addMember	Class.ftl	Add a member (attribute or method) to a Java class.
ClassConstructor:addAnnotations	Constructor.ftl	Add an annotation to a Java constructor.
EnumContent:addComment	Enum.ftl	Add a comment to a Java enumeration.
InterfaceContent:addComment	Interface.ftl	Add a comment to a Java interface.
InterfaceContent:addImports	Class.ftl	Add additional imports to a Java interface.
InterfaceContent:addAnnotations	Class.ftl	Add annotations to a Java interface.
InterfaceContent:addMember	Class.ftl	Add a member (attribute or method) to a Java interface.

ClassMethod:addAnnotations	Method.ftl	Add annotations to a Java method.
ConcreteController:addToolBar-Button	GetToolBar-Components-MethodBody.ftl	Add a widget to the ToolBar.

Table H.1: List of hook points in the MontiDExcode generator.

Appendix I

Curriculum Vitae

Personal Data

Family Name:	Roth
First Name:	Alexander
Date of Birth:	25.03.1986
Place of Birth:	Ujar (Russia)
Nationality:	German

Academic Employment

10/2012 - 09/2017:	RWTH Aachen University: Research Assistant and Team Leader of the Digitalization Group

Education

10/2012 - 11/2017:	RWTH Aachen University: Ph.D. studies in Software Engineering
10/2010 - 09/2012:	RWTH Aachen University: Computer Science studies Master of Science in Computer Science
10/2007 - 09/2010:	RWTH Aachen University: Computer Science studies Bachelor of Science in Computer Science
07/2006	Albert-Einstein-Gymnasium Sankt Augustin: German Abitur

List of Figures

Listings

List of Tables

Related Interesting Work from the SE Group, RWTH Aachen

Agile Model Based Software Engineering

Agility and modeling in the same project? This question was raised in [Rum04]: "Using an executable, yet abstract and multi-view modeling language for modeling, designing and programming still allows to use an agile development process." Modeling will be used in development projects much more, if the benefits become evident early, e.g with executable UML [Rum02] and tests [Rum03]. In [GKRS06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum12] and [Rum16], 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] 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]. 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 reliability of adaprivity through models at runtime, which will allow developers to delay design decisions to runtime adaptation.

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 based on the MontiCore language workbench [KRV10, GKR+06, GKR+08]. In [KRV06], we discuss additional roles necessary in a model-based software development project. In [GKRS06] we discuss mechanisms to keep generated and handwritten code separated. In [Wei12] demonstrate how to systematically derive a transformation language in concrete syntax. To understand the implications of executability for UML, we discuss needs and advantages of executable modeling with UML in agile projects in [Rum04], 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)

Starting with an 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 two books [Rum16] and [Rum12] 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" [BCGR09a], [BCGR09b], [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 [MRR11c] and objects diagrams [MRR11d] or the consistency of both kinds of diagrams [MRR11e]. We also apply these concepts to activity diagrams [MRR11b] which allows us to check for semantic differences

of activity diagrams [MRR11a]. The basic semantics for ADs and their semantic variation points is 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+98] and how to use modeling in agile development projects [Rum04], [Rum02]. The question 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 [EFLR99], [FELR98] and [SRVK10].

Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use, but need appropriate tooling. The MontiCore language workbench [GKR+06, KRV10, Kra10, GKR+08] allows the specification of an integrated abstract and concrete syntax format [KRV07b] for easy development. New languages and tools can be defined in modular forms [KRV08, GKR+07, Völ11] and can, thus, easily be reused. [Wei12] presents a tool that allows to create transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11]. A successful application has been carried out in the Air Traffic Management domain [ZPK+11]. Based on the concepts described above, meta modeling, model analyses and model evolution have been discussed in [LRSS10] and [SRVK10]. DSL quality [FHR08], instructions for defining views [GHK+07], guidelines to define DSLs [KKP+09] and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

Software Language Engineering

For a systematic definition of languages using 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+15]. As said, the MontiCore language workbench provides techniques for an integrated definition of languages [KRV07b, Kra10, KRV10]. In [SRVK10] we discuss the possibilities and the challenges using metamodels for language definition. Modular composition, however, is a core concept to reuse language components like in MontiCore for the frontend [Völ11, KRV08] and the backend [RRRW15]]. Language derivation is to our believe a promising technique to develop new languages for a specific purpose that rely on existing basic languages. How to automatically derive such a transformation language using 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+15a, HHK+13], where a delta language is derived from a base language to be able to constructively describe differences between model variants usable to build feature sets.

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] for semantics. Furthermore, we built a concrete tooling infrastructure called MontiArc [HRR12] for architecture design and extensions for states [RRW13b]. MontiArc was extended to describe variability [HRR+11] using deltas [HRRS11, **?**] and evolution on deltas [HRRS12]. [GHK+07] and [GHK+08] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants. [MRR14] provides a precise technique to verify consistency of architectural views [Rin14, MRR13] against a complete architecture in order to increase reusability. Co-evolution of architecture is discussed in [MMR10] and a modeling technique to describe dynamic architectures is shown in [HRR98].

Compositionality & Modularity of Models

[HKR+09] motivates the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07] and algebraically underpinned in [HKR+07]. Semantic and methodical aspects of model composition [KRV08] led to the language workbench MontiCore [KRV10] that can even be used to develop modeling tools in a compositional form. A set of DSL design guidelines incorporates reuse through this form of composition [KKP+09]. [Völ11] examines the composition of context conditions respectively the underlying infrastructure of the symbol table. Modular editor generation is discussed in [KRV07a]. [RRRW15] applies compositionality to Robotics control. [CBCR15] (published in [CCF+15]) 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]. It allows to describe additional information for model elements in separated documents, facilitates reuse, and allows to type 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]. An extended version especially suited for the UML is given in [BCGR09b] and in [BCGR09a] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied to class diagrams in [CGR08]. To better understand the effect of an evolved design, detection of semantic differencing as opposed to pure syntactical differences is needed [MRR10]. [MRR11a, MRR11b] encode a part of the semantics to handle semantic differences of activity diagrams and [MRR11e] compares class and object diagrams with regard to their semantics. In [BR07], a simplified mathematical model for distributed systems based on black-box behaviors of components is defined. Meta-modeling semantics is discussed in [EFLR99]. [BGH+97] discusses potential modeling languages for the description of an exemplary object interaction, today called sequence diagram. [BGH+98] 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 on class and object diagrams in [MRR11e] 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 evolution and related issues in greater detail.

Evolution & Transformation of Models

Models are the central artifact 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. Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04], refinement [PR99, KPR97, PR94], refactoring [Rum12, PR03], translating models from one language into another [MRR11c, Rum12] and systematic model transformation language development [Wei12]. [Rum04] 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], refining pipe-and-filter architectures is explained in [PR99]. Refactorings of models are important for model driven engineering as discussed in [PR01, PR03, Rum12]. Translation between languages, e.g., from class diagrams into Alloy [MRR11c] allows for comparing class diagrams on a semantic level.

Variability & 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] 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 [HRR+11, HRR+11] and to Delta-Simulink [HKM+13]. Deltas can not only describe spacial variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK+13] and [HRW15] describe an approach to systematically derive delta languages. We also apply variability to modeling languages in order to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02]. Furthermore, we specified a systematic way to define variants of modeling languages [CGR09] and applied this as a semantic language refinement on Statecharts in [GR11].

Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12] are complex, distributed systems which control physical entities. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12] and autonomous driving [BR12a] 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 for the European airspace [ZPK+11]. A component and connector architecture description language suitable for the specific challenges in robotics is discussed in [RRW13b, RRW14]. Monitoring for smart and energy efficient buildings is developed as Energy Navigator toolset [KPR12, FPPR12, KLPR12].

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, BCGR09b, BCGR09a], (2) understanding the refinement [PR94, RK96, Rum96] and composition [GR95] of statemachines, and (3) applying statemachines for modeling systems. In [Rum96] 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 [BR07]. We apply these techniques, e.g., in MontiArcAutomaton [RRW13a, RRW14] as well as in building management systems [FLP+11].

Robotics

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 composition and interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers broad propagation of robotics applications. The MontiArcAutomaton language [RRW13a] extends ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13b, RRW14, RRRW15] that perfectly fit Robotic architectural modelling. The LightRocks [THR+13] framework allows robotics experts and laymen to model robotic assembly tasks.

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 requirements management that connects requirements with features in all phases of the development 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]. [HKM+13] describes a tool for delta modeling for Simulink [HKM+13]. [HRRW12] discusses means to extract a well-defined Software Product Line from a set of copy and paste variants. [RSW+15] describes an approach to use model checking techniques to identify behavioral differences of Simulink models. Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12a, BR12b], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in development and evolution of autonomous car functionality, and thus enables us to develop software in an agile way [BR12a]. [MMR10] gives an overview of the current 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. As tooling infrastructure, the SSElab storage, versioning and

management services [HKR12] are essential for many projects.

Energy Management

In the past years, it became more and more evident that saving energy and reducing CO2 emissions is an important challenge. Thus, energy management in buildings as well as in neighbourhoods 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 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 to validate it against the monitored values [FLP+11]. We show how our data model, the constraint rules and the evaluation approach to compare sensor data can be applied [KLPR12].

Cloud Computing & Enterprise Information Systems

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, to lower the barrier for web-based innovations and to increase the efficiency and cost-effectiveness of web development [KRR14]. Application classes like Cyber-Physical Systems and their privacy [HHK+14, HHK+15b], Big Data, App and Service Ecosystems bring attention to aspects like responsiveness, privacy and open platforms. Regardless of the application domain, developers of such systems are in need for robust methods and efficient, easy-to-use languages and tools [KRS12]. We tackle these challenges by perusing a model-based, generative approach [NPR13]. 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 own development platforms. New services, e.g., collecting data from temperature, cars etc. can now easily be developed.

[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. I, I

[BCGR09a] 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. I, I, I

[BCGR09b] 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. I, I, I

[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. I, I

[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. I, I, I

[BGH+97] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. In *Object-oriented Behavioral Semantics Workshop (OOPSLA'97)*, Technical Report TUM-I9737, Germany, 1997. TU Munich. I

[BGH+98] 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. I, I

[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. I, I

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007. I, I, I, I

[BR12a] 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. I, I

[BR12b] 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. I

[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. I, I

[CCF+15] Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe, editors. *Globalizing Domain-Specific Languages*, LNCS 9400. Springer, 2015. I, I

[CEG+14] Betty Cheng, Kerstin Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi Müller, Patrizio Pelliccione, Anna Perini, Nauman Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In *Models@run.time*, LNCS 8378, pages 101–136. Springer, Germany, 2014. I

[CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany, 2008. I, I

[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. I, I

[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. I, I

[FELR98] Robert France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19(7):325–334, November 1998. I

[FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008. I, I, I

[FLP+11] 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. I, I

[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. I, I, I

[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. I, I, I

[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. I, I, I

[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. I

[GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TU Munich, Germany, July 1996. I, I

[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. I, I, I

[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. I

[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. I, I, I

[GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, and Martin Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Modellierung 2006 Conference*, LNI 82, Seiten 67–81, 2006. I, I

[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. I, I

[GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TU Munich, Germany, October 1995. I

[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. I, I, I, I

[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. I, I, I

[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. I, I

[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. I, I

[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. I

[HHK+15a] 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. I

[HHK+15b] 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. *Future Generation Computer Systems*, 56:701–718, 2015. I

[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. I, I

[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. I

[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. I, I

[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. I, I

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004. I

[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. I

[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. I, I

[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. I, I

[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. I

[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. I, I

[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. I, I

[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. I, I

[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. I

[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. I, I

[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. I, I

[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. I, I

[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. I, I, I

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering.* Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, März 2010. I, I

[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. I

[KRR14] Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe. *Trusted Cloud Computing.* Springer, Schweiz, December 2014. I

[KRS12] Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Sys-

tems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, Seiten 113–116. VDI Verlag, 2012. I, I

[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. I, I

[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. I, I

[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. I, I

[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. I, I, I

[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. I, I, I, I

[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. I, I, I, I

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, May 2010. I, I, I

[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. I

[MRR11a] 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. I, I

[MRR11b] 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. I, I

[MRR11c] 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. I, I

[MRR11d] 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. I

[MRR11e] 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. I, I

[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. I

[MRR14] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *Software Engineering Conference (ICSE'14)*, pages 95–105. ACM, 2014. I

[NPR13] 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. I

[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. I, I

[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. I, I

[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. I, I

[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. I

[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. I, I

[Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, 2014. I

[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. I

[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. I

[RRRW15] 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. I, I, I

[RRW13a] 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. I, I

[RRW13b] 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. I, I, I

[RRW14] 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. I, I, I

[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. I

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme.* Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996. I

[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. I, I, I

[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. I, I

[Rum04] 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. I, I, I, I

[Rum11] Bernhard Rumpe. *Modellierung mit UML, 2te Auflage.* Springer Berlin, September 2011. I

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring, 2te Auflage.* Springer Berlin, Juni 2012. I, I, I, I, I

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods.* Springer International, July 2016. I, I, I

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P.* Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012. I, I, I

[SRVK10] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Meta-

modelling: 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. I, I, I

[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. I

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen.* Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011. I, I, I

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen.* Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012. I, I, I, I

[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. I, I