Institute of Software Engineering
Software Quality and Architecture

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master’s Thesis

Redesigning the Hamster Simulation

Mario Fuksa

Course of Study:	Informatik
Examiner:	Prof. Dr.-Ing. Steffen Becker
Supervisor:	Prof. Dr.-Ing. Steffen Becker
Commenced:	October 15, 2020
Completed:	April 15, 2021

Abstract

Teaching programming can be performed in many different ways, such as focusing on object-oriented concepts in first place combined with mini-worlds like the hamster simulator. At the University of Stuttgart, such a solution is provided with the focus on teaching Java. But now the need to teach C++ in a similar way is also intended. To solve this necessity, a model-driven solution is proposed to model mini-worlds and generate code into multiple programming languages like Java or C++.

The proposed solution covers requirements based on the existing approaches relevant for teaching programming at the University of Stuttgart at the Institute of Software Engineering. Based on these requirements, a modeling environment is designed which provides a framework part and a concrete mini-world simulator part. Technically, the modeling environment is based on the Eclipse platform and makes use of research related tooling for input modeling, model-to-model transformations and code generation. Generated simulators are based on a modular architecture, which enables high automation for tests and independence of concrete third-party frameworks. Further, the interface provided for students is based on object-oriented principles and contract-based design, including formalized pre- and postconditions defined for commands. By providing a meta-model to define mini-worlds in a generic way, the proposed solution can be adapted for modeling of different mini-worlds. In addition, a code generator is developed to transform adjusted intermediary models to concrete source code. By achieving that most complexity is handled by model-to-model transformations, this allows also to adapt the solution for further programming languages.

Finally, different aspects for the proposed solution are evaluated. On the one hand, the combination of object-oriented teaching concepts with model-driven software development is evaluated. On the other hand, the use of existing ideas such as the generation of graph transformations at Fujaba or technologies such as Henshin is discussed. The functionality of the solution is shown by adapting it to the Java and C++ programming languages. Furthermore, another mini-world is adapted in addition to the hamster simulator with *Kara the ladybug*.

Kurzfassung

Das Lehren von Programmierung kann auf viele verschiedene Arten erfolgen, wie beispielsweise durch die Fokussierung auf objektorientierte Konzepte in Kombination mit Mini-Welten wie dem Hamster-Simulator. An der Universität Stuttgart wird eine solche Lösung mit dem Fokus auf das Lehren von Java angeboten. Aufbauend soll nun auch C++ in ähnlicher Weise gelehrt werden. Um diese Notwendigkeit zu erfüllen, wird eine modellgetriebene Lösung vorgeschlagen, um Mini-Welten zu modellieren und Code in mehrere Programmiersprachen wie Java oder C++ zu generieren.

Die vorgeschlagene Lösung deckt Anforderungen ab, die auf den bestehenden Ansätzen basieren, welche für die Lehre der Programmierung an der Universität Stuttgart am Institut für Softwaretechnik relevant sind. Basierend auf diesen Anforderungen wird eine Modellierungsumgebung entworfen, die einen Framework-Teil und einen konkreten Miniwelt-Simulator-Teil bereitstellt. Technisch basiert die Modellierungsumgebung auf der Eclipse-Plattform und nutzt forschungsnahe Werkzeuge für die Eingabe-Modellierung, Modell-zu-Modell-Transformationen und Codegenerierung. Die generierten Simulatoren basieren auf einer modularen Architektur, die eine hohe Testautomatisierung und die Unabhängigkeit von externen Frameworks ermöglicht. Darüber hinaus basiert die für Studierende bereitgestellte Schnittstelle auf objektorientierten Prinzipien und vertragsbasiertem Design, einschließlich formalisierter Vor- und Nachbedingungen für Kommandos. Durch die Bereitstellung eines Meta-Modells zur generischen Definition von Mini-Welten kann die vorgeschlagene Lösung für die Modellierung verschiedener Mini-Welten angepasst werden. Zudem wird ein Code-Generator bereitgestellt, um geeignete Zwischenmodelle in konkreten Quellcode zu transformieren. Durch die Tatsache, dass die meiste Komplexität durch Modell-zu-Modell-Transformationen gehandhabt wird, ermöglicht dies die Anpassung der Lösung auch für weitere Programmiersprachen.

Schließlich werden verschiedene Aspekte für die vorgeschlagene Lösung evaluiert. Einerseits wird das Zusammenspiel aus objektorientierte Lehrkonzepte mit der modellgetriebenen Softwareentwicklung bewertet. Andererseits wird der Einsatz bestehender Ideen wie die Generierung von Graph-Transformationen bei Fujaba oder Technologien wie Henshin diskutiert. Die Funktionsweise der Lösung wird durch die Adaptierung auf die Programmiersprachen Java und C++ gezeigt. Außerdem wird neben dem Hamster-Simulator mit *Kara der Marienkäfer* eine weitere Mini-Welt adaptiert.

Acknowledgements

At this point I would like to thank my supervisor Prof. Dr.-Ing. Steffen Becker for making this master thesis possible. The redesign of the hamster simulator with a model-driven approach was a lot of fun. Through this topic I was able to gain a huge amount of inspiration for future work. The regular meetings were very valuable and also a pleasure to have. Without your help, the result would certainly not have been as good. Especially in the Corona Pandemic, the meetings have also always provided a dose of motivation and energy.

Another special thanks goes to my fiancee Vanessa, who always supported me during this work. In addition, I would like to thank the people who read my thesis for proofreading.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

AGG Attributed Graph Grammar System. 17

ALF Action Language for Foundational UML. 19

API Application Programming Interface. 1

AST Abstract Syntax Tree. 16

CASE Computer Aided Software Engineering. 24

CDT C/C++ Development Tooling. 48

DMM Dynamic Meta Modeling. 2

DSL Domain Specific Language. 12

EBNF Extended Backus-Naur Form. 16

EMF Eclipse Modeling Framework. 2

EMOF Essential Meta Object Facility. 10

fUML Foundational UML. 19

IDE Integrated Development Environment. 2

ISTE Institute of Software Technology. 1

JML Java Modeling Language. 28

JVM Java Virtual Machine. 16

LTS Labelled Transition System. 82

MDSD Model-driven Software Development. 2

MOF Meta Object Facility. 10

MPW Mini-Programming-World. 1

MWE2 Modeling Workflow Engine 2. 31

OCL Object Constraint Language. 5

OMG Object Management Group. 9

PSE Programming and Software Engineering. 1

QVT Query View Transformation. 12

QVT-O QVT-Operational. 2

SDK	Software Development Kit. 48
SDL	Simple DirectMedia Library. 32
SDM	Story Driven Modeling. 24
TGG	Triple Graph Grammars. 13
UI	User Interface. 20
UML	Unified Modeling Language. 9
URI	Uniform Resource Identifier. 65
XMI	XML Metadata Interchange. 15
XML	Extensible Markup Language. 12
XSLT	XSL Transformation. 12

1 Introduction

These days, computers can be used to solve many different problems. To translate requirements into executable code, programmers have to create programs, which consist of sequences of statements [BB14].

For many programmers, their programming skills are thought in programming courses, where teachers have to select proper methods and contents to teach. But teaching programming can be done in many different approaches, e.g. by focusing on a certain programming paradigm like the object-oriented programming from the start. Additionally, there are several challenges for teachers, since students have to be prepared for real challenges, while core principles have to be thought for durability. Simultaneously, the quality of exercises is critical to keep students motivated [PM06].

One practical way is to teach the object-oriented programming paradigm by using gamification scenarios, which can be combined with the *Outside-In* approach. This approach targets to teach concepts of object-oriented languages including abstraction and interface design first. Later, standard low-level concepts like data-structures and algorithms are introduced [Mey09].

Based on this idea, at the University of Stuttgart the introductory course Programming and Software Engineering (PSE) is offered for students to learn programming [BBF20]. On the one hand, the *hamster simulator* originally developed by Boles is used as a concrete gamification model, which belongs to the class of Mini-Programming-Worlds (MPWs) [BB14]. On the other hand, the didactics is combined with the Outside-In approach to profit from the benefits of focusing on object-orientation and well-defined Application Programming Interface (API) design.

1.1 Problem Statement

The original hamster simulator is not developed for teaching the concepts of Outside-In in the first place. Therefore a re-implementation by the Institute of Software Technology (ISTE) at the University of Stuttgart has been performed in the past [BBF20]. This re-implementation for the PSE course will be called *PSE-Simulator*, while the original hamster simulator will be called *Boles-Simulator* in this thesis. One major advantage of the PSE-Simulator is that the client programs in an external IDE like Eclipse or IntelliJ, which enables full debugging, auto-completion, static-code-analysis and further tooling support. This allows clients to further use the latest Java version, since there are no third-party dependencies which could make the migration difficult.

While the PSE-Simulator re-implementation has a focus on the Java programming language and is successfully applied for teaching, there is now a need to teach other programming languages such as C++ as well. A first approach has been tried to wrap the Java-based simulator with a C API and hence enable C++ programs to interact with the existing PSE-Simulator. However, this approach has some weaknesses and leads to unsatisfactory experiences.

Therefore the overall problem statement for this thesis is to develop another solution to support reuse teaching concepts based on the hamster simulator for teaching C++. This leads to a redesign of the PSE-Simulator which is called *Proposed-Simulator* in context of this thesis. Besides the support of multiple programming languages, the Proposed-Simulator shall include the basic design goals of the PSE-Simulator like Outside-In concepts or Integrated Development Environment (IDE) flexibility. Further, the solution shall support the adaption for additional MPWs like *Kara the ladybug*, to enable more variation while teaching programming.

1.2 Solution Approach

The motivation of this thesis is to design a portable system, where MPWs like the hamster simulator are natively available in other programming languages like C++. As the central idea, the proposed approach will be based on Model-driven Software Development (MDSD). It combines advantages of the Boles-Simulator and the PSE-Simulator using a multi-language approach, which is designed for Outside-In concepts. This native multi-language approach provides full debugging support and full transparency for the client, by avoiding technical boundaries which are introduced by API-wrappers.

As a more concrete MDSD approach, the Dynamic Meta Modeling (DMM) will be applied for modeling of dynamic aspects like moving the hamster in the territory. To realize this, research related tools based on the Eclipse Modeling Framework (EMF) will be used. This includes Henshin to model graph transformation rules, Xtext to create a query language, QVT-Operational (QVT-O) for model-to-model transformations and Xpand to finally generate source code.

Further, Boles developed another tool called *Solist* with the support of creating arbitrary MPWs [Bol]. The solution approach of this thesis will make reuse of the meta-model used in Solist to support also the modeling of multiple MPWs.

To summarize the contribution of this thesis, a MDSD framework will be created to model different MPWs for multiple programming languages like Java or C++.

1.3 Thesis Structure

The thesis is structured as follows:

Chapter 2 – Foundations: Covers relevant foundations used to develop the proposed solution for this work.

Chapter 3 – Related Work: Gives an overview of related works, which are relevant for this thesis and are used as important orientations.

Chapter 4 – Concept: Outlines the overall concepts and approach used for developing the proposed solution.

Chapter 5 – Modeling Workflow: Describes the modeling workflow and its realization in more detail.

Chapter 6 – Evaluation: Evaluates the goals and results of this work by focusing on several research questions.

Chapter 7 – Conclusion: Concludes the final results and gives ideas for improvements and extensions for future works.

2 Foundations

This chapter outlines the relevant foundations of this work. Figure 2.1 gives a brief overview of the concepts and technologies used. Additionally, the relations of the foundations are depicted. First the main topic *Teaching Programming* is described in Section 2.1 on the following page. The focus of this section lies on the concepts of the *Outside-In* approach based on the book *Touch of Class* by Meyer [Mey09]. Especially the concepts of *contracts* are relevant in this work, which are also described in more detail.

The second main topic *MDSD* is handled in Section 2.2 on page 8, where the concepts of *meta-models* and *model transformations* are described. Concepts and related technologies like the EMF for describing meta-models, *Xtext* to define a concrete syntax or the *Object Constraint Language (OCL)* to specify *static semantics* are illustrated. Additionally, *QVT-O* as a model transformation language is introduced, which is intensively used in this work. Another important foundation is about *dynamic semantics*, where *graph transformations* and the used tool environment *Henshin* are described in detail.

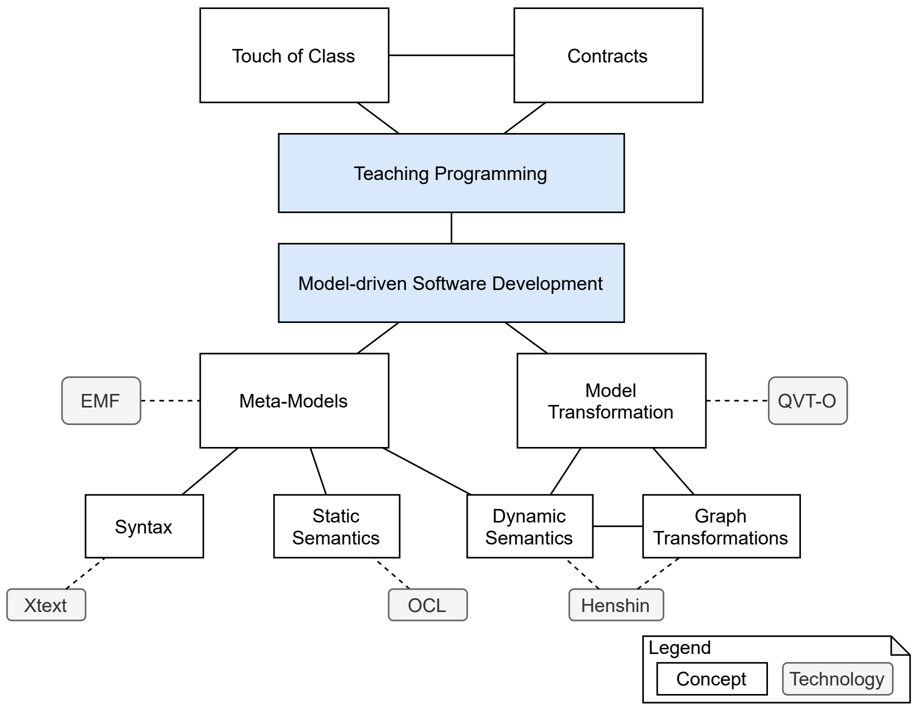

Figure 2.1: Overview of relevant foundations

2.1 Teaching Programming

This section outlines the related foundations of the programming course which is the context of this work. It is fundamentally based on the Outside-In approach [Mey09]. The preceding work of the approach described in the book is also reported in a paper of Pedroni and Meyer [PM06].

2.1.1 Outside-In Approach by Touch of Class

Pedroni and Meyer stated in their paper that traditional teaching approaches do not properly address the introductory of programming [PM06]. They observed that teachers often use object-oriented programming languages like Java or C++ to teach programming, but their focus lies usually on the low-level concepts like algorithms and data-structures [Mey09]. Their approach is called *Outside-In*. It is an inverted one compared to the traditional practice. The term *Objects-First* is used as a synonym in this thesis. Key aspects in this approach are the object-orientation, *Design by Contract* and the reuse of a software framework developed for teaching. Students are learning concepts like components and how to program from the perspective of a consumer first, where the design of APIs and documentation plays an important role. Later on, they will learn the role of the producer side and the relating techniques to program the implementation details [PM06].

Pedroni and Meyer identified the following six challenges when teaching introductory programming [PM06]:

- Durability of skills: Teachers shall educate future software professionals in a way that their skills are more durable. Globalization leads to massive outsourcing and the option that cheaper programmers are available, which are skilled in a required immediately applicable technology. This brings responsibility for teachers to focus on teaching durable and long-term relevant skills to allow students to be demanded even when short-term technology skills become obsolete.
- Precisely define what to teach: Programming can be seen as an elementary form which is exposed to a large part of the population. Hence it is important that it must be precisely defined which contents shall be taught in a programming education.
- Diversity of knowledge: Through the growing impact of software, the diversity of backgrounds of the students grows. The challenge of teaching programming is to find the right approach. On the one side, it shall not be too difficult for those who have barely touched the world of computers. On the other side, it must also be interestingly enough for those with extensive programming experience. With the use of components from the start the novices have simple and more abstract interfaces which are more understandable to use. The more advanced students can optionally dive into the internals of a component to learn from them.
- Quality of examples: People from the ”Nintendo generation” and later are unlikely to be impressed by small and too abstract examples which are traditionally used. To improve the quality of examples, libraries providing advanced graphics, multimedia and interaction capabilities can be introduced.

- Teaching the real challenges: Teaching in the small is often not sufficient to prepare students for professional software development with large systems. While combining teaching and practice is one approach to tackle this challenge, it may also be possible that students are faced with large programs even in the university context.
- Introduce advanced principles: The sixth issue is how to teach also more advanced, essential principles without disconnecting from students. As an example, students can be taught how to work with a well-defined interface of components and principles such as Design by Contract.

To overcome the challenges previously described, Pedroni and Meyer defined multiple principles which they follow in their introductory programming course [PM06]:

- Objects first: From the beginning they use object-oriented concepts in their course. They argue that this approach is natural, especially for the introduction, since classes reflect things which are familiar to the students.
- Components: Students get access to multiple existing, feature-rich libraries to learn how to deal with components. This way they can produce impressive applications from the start with only few lines of code. On the one side this teaches the importance of reuse and abstraction, while on the other side it catches the interest how libraries work.
- Abstraction and contracts: With contracts, as an important concept, the specification of routines can be formalized in a solid way. They are implemented with preconditions, postconditions and class invariants which help students to learn to correctly call routines as a conform consumer.
- Order of topics: Like mentioned before, in their Outside-In approach first *outer* structures like classes, interfaces, objects and features are taught. This is followed by the *inner* structure consisting of building blocks such as variables, assignments or control structures.
- Formality: By including a certain degree of formality, they teach in a durable way that mathematical foundations are also important for practical programming. Examples are loop invariants and the concepts of Design by Contract which seems to be the right portion of formality.
- The source framework: They introduce a base framework to let students build their programs. It provides an immediately familiar context for the students and a rich base for interesting algorithms and data-structure examples. Further, multi-media and advanced graphics are included.

The reference course PSE [BBF20], which uses the hamster simulator, is also based on the Outside-In approach. Therefore the concepts are important to be considered in the solution of this work as well.

2.1.2 Contracts

Based on the Outside-In approach the terminology for defining contracts is described in this section. This terminology is used in this work to model behavior and hence represents an important aspect.

Operations used to deal with an object are classified into *queries* and *commands*. With queries information of an object is *accessed* and returned in a side-effect free manner. In contrast, commands *modify* the state of an object. In addition, operations can be extended by *contracts* which precisely describe what is permitted. *Preconditions* are imposed to all clients and restrict the state or arguments when invoking the operation. The client calling an operation has to ensure that the required preconditions hold. In contrast, *postconditions* are used to describe the ensured properties which hold after successfully processing an operation. While postconditions are processed at the end of an operation, they may refer to a value which is present at the entry point. This is performed with *old* expressions. Further, *class invariants* are defined on the context of a class and must hold as soon as an object is created. They are checked at any time before or after the invocation of an operation [Mey09].

2.2 Model-Driven Software Development

This section describes the MDSD approach in detail, which is the characteristic method for software development used in this thesis.

2.2.1 Meta Modeling

The definition of Stachowiak can be used to define the word *model*. According to his definition, a model is a representation of the reality, which is reduced to the essentials, depending on the modeling context. He defines three characteristics which can be identified on a model [Sta73]:

1. Homomorphism: Models are representations of natural or artificial originals. Statements made on a model must also hold for the real entity.
2. Abstraction: The model is a simplified representation of the real entity. Not every detail will be retained.
3. Pragmatics: A model is created with a specific intent. It fulfills a replacement of one or multiple originals.

According to Kühne a meta-model is a model of another model [Küh06]. Stahl et al. state that it is important to formalize the structure of a domain which results in a meta-model. The structure can be divided into the following parts [VSB+13]:

- Concrete Syntax: Represents a concrete form of a language like the textual syntax of Java. It is a realization of an abstract syntax, where it is possible that multiple concrete ones are related to one abstract syntax [VSB+13]. The concrete syntax can be represented in a graphical or a textual form.

- Abstract Syntax: Defines how the structure of a language looks like. It abstracts its relating concrete syntax and omits details used in the concrete syntax like keywords. Often a parser instantiates the abstract syntax to represent e.g. the textual program code as an object tree in the program's memory [VSB+13].
- Static Semantics: Describes the validation criteria for a language. It plays an important role in the context of MDSD since it can detect modeling errors and enforce that models are valid. As an example, the OCL can be used to specify constraints on the abstract syntax of a language. These constraints are ideally checked at model time [VSB+13].
- Dynamic Semantics: Specifies the meaning of the meta-model elements. While static semantics define aspects of a model at a specific point in time, the dynamic semantics define how individual elements in the model change over time. The Unified Modeling Language (UML) shows some important distinctions for dynamic semantics. For example, the operational behaviors define how the state of a class changes by including input and output parameters and optionally including pre- and postconditions. Other distinctions are property default values or the semantics, which are activated when an object is created [Obj17]. The related approach of DMM to model dynamic semantics is described in Section 2.2.2.

Besides the structural view of models, the hierarchical relationships between models can be also considered which is shown in Figure 2.2.

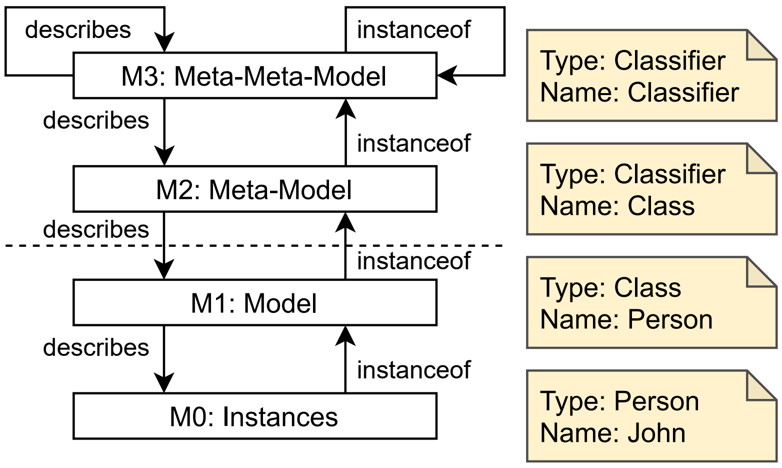

Figure 2.2: Meta-layers defined by the OMG [VSB+13]

For this, the Object Management Group (OMG) defined a hierarchy of models and their meta-models, where it holds that every model has a meta relationship to a meta-model. In the M1 layer the model is defined which is usually used to generate source code. One level below, the M0 layer contains instances which are instantiated by their relating class. The instances represent real entities like a person named John. For the attributes defined in their relating class concrete values are defined. Classes like `Person` of the instances like John are described in the meta-layer M1 and they define their attributes, operations or relationships. Above the dotted line the meta-layers are shown. The layer M2 represents the meta-model of the model in M1 and in this layer concepts are defined, which are used in the M1 layer to describe instances in the M0 layer. For example, a `Class` is an instance of the meta-class `Classifier`. The upper layer M3 defines a meta-meta-model which is used to describe the M2 layer and its meta-model. Like depicted in the figure, M3 also recursively instantiates and describes itself. This means that the meta-meta-model is defined in a way that its concepts like `Classifier` can be used to describe themselves [VSB+13]. While theoretically more layers could be defined, for most systems the four layers are sufficient [Obj19].

The OMG defines the widely used Meta Object Facility (MOF) where a meta-meta-model is defined which is the base of UML. A reduced version of the complete MOF is the Essential Meta Object Facility (EMOF) which has been designed to be aligned with object-oriented programming languages. As a primary goal it shall allow simple meta-models to be defined [Obj19]. The meta-meta-model used in the EMF has influenced the specification of EMOF and they closely equal each other [SBMP08].

2.2.2 Dynamic Meta Modeling

DMM is a meta-modeling-based approach which can be used to specify dynamic semantics of models. The approach was proposed by Engels et al. in their paper to propose a graphical approach to enrich UML behavior diagrams by operational semantics [EHHS00]. One goal of DMM is to reach highly understandable semantic models, which are precisely enough for formal analysis [ESW07].

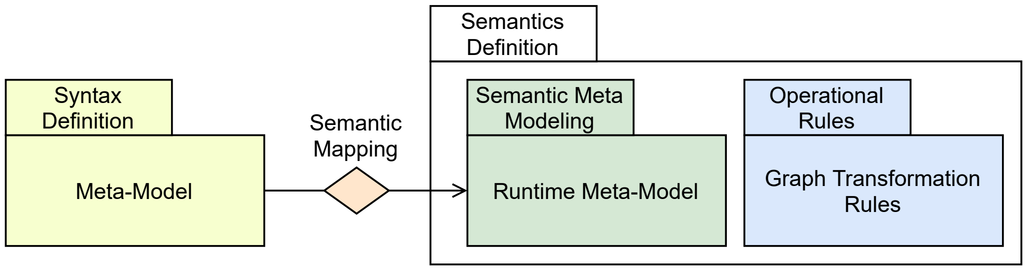

Figure 2.3: Overview of the DMM approach (based on [SE10])

Figure 2.3 gives an overview over the core components of the DMM approach. A requirement for applying DMM is a static meta-model which describes the abstract syntax of a model. Based on this static meta-model an extended runtime meta-model can be derived. Through a semantic mapping, elements of the static meta-model can be mapped to elements of the runtime meta-model. Additionally operational rules are defined by using graph transformation rules. These rules act as transitions and describe how instances of the runtime model are modified to result in other instances of the runtime model. Finally, this specification can be used to compute a transition system which precisely represents the operational behavior of models [ESW07]. In addition to a visual representation of the behavior, DMM allows further analysis like testing the transitions of certain model instances [BSE10].

The ideas of DMM are representative to the approach of modeling the operational behavior of MPWs in this work. For example, the operational rule to pick a grain in the hamster simulator is modeled as a graph transformation and operates on a concrete instance of the runtime model. As part of this exemplary transition, a grain is removed from the territory and added to the hamster's mouth. Therefore, DMM can be seen as the general approach to model dynamic semantics used in this work.

2.2.3 Model Transformations

While transformations in general are a basic concept in software development and every computation can be viewed as a data transformation, model transformations can be viewed as a form of meta-programming where the semantics of meta-data are handled. Model transformations are a key aspect of MDSD and they usually operate on object-oriented representations of models. There are the following applications for which model transformations are primarily used [CH06]:

- generate lower-level models or code from higher-level models
- perform a mapping or synchronization of models on the same abstraction level
- create query-based views on a system
- perform model refactorings
- to reverse engineer higher-level models from lower-level ones

In Figure 2.4 the basic concepts of a model transformation are shown. Basically, there is a *source model* which is transformed by a *transformation engine* into a *target model*. The transformation engine uses for the transformation a *transformation definition*, which refers to the meta-models of the source and target model. Source and target models might be the same, or it might also be the case that there are multiple models. The former case is called in-place transformation as well.

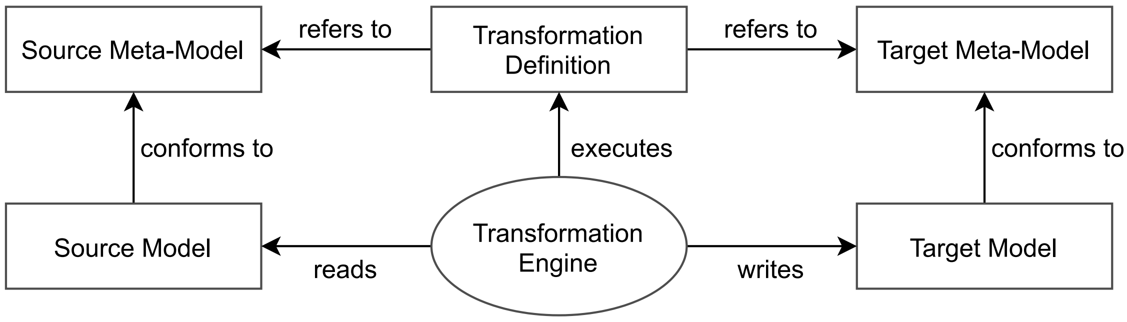

Figure 2.4: Basic concepts of model transformation [CH06]

Model transformations can be distinguished into two major categories *model-to-text* and *model-to-model*. For the former category the following approaches are identified:

- Visitor-Based: This approach uses a visitor mechanism to traverse the input model and then write text to the output stream.
- Template-Based: Often tools use this approach where a template containing the target text and parts of meta-code is processed to generate the final text.

The latter major category contains approaches where both the source and the target instances are based on a meta-model [CH06]:

- Direct-Manipulation: This kind of approaches is very basic. It uses an internal model representation which is modified by some API. Usually transformation rules, scheduling, tracing and other facilities have to be implemented from scratch.

- Structure-Driven: Here two phases are distinguished. The first phase is used to create the hierarchical structure of the target model. In the second phase, attributes and references in the target are set. Users of a structure-driven approach usually do not have to deal with scheduling and application strategies.
- Operational: Similar to a direct manipulation approach a model instance is directly manipulated using operations. But in contrast for operational approaches more tooling support is given e.g. by tracing capabilities or by extending the meta-model facilities like querying elements.
- Template-Based: Here model templates are used which embeds meta-code into variable parts of the target instances. One possible realization is to embed concrete syntax like source code or OCL expressions into annotations of model elements. Different kinds of annotations might be used like conditions, iterations or expressions to reach the possibility to embed imperative logic in the meta-language.
- Relational: This group is based on declarative approaches and mainly uses mathematical relations. Generally, they can be viewed as an approach where given constraints on models shall be solved. One important property of relational approaches is that they are side-effect-free, where non-executable specifications like relations or mapping rules are executed by the transformation engine.
- Graph-Transformation-Based: In this category graph transformations are used, which operate on typed, attributed and labeled graphs. Since they play an important role in this work they will be described in more detail in the next subsection.
- Hybrid: If approaches combine different techniques described in the previous categories, they are classified as hybrid. An example is Query View Transformation (QVT) where the three components named *Relations*, *Operational mappings* and *Core* are combined.
- Others: There are also other approaches like XSL Transformation (XSLT) which can be classified as *term rewriting* using a functional language. XSLT is a standalone technology which performs on Extensible Markup Language (XML) documents. Another approach in this category is the application of meta-programming to perform model transformations, where e.g. a Domain Specific Language (DSL) is embedded in a meta-programming language.

In this work the operational, template-based and graph-based model-to-model approaches are mainly used.

2.2.4 Graph Transformations

This category of transformations is based on the theoretical work on graph transformations [CH06]. Graphs consist of nodes and edges, where nodes are connected by edges. Variants are that they can be directed or undirected, labeled or unlabeled, attributed or not attributed. In addition, graphs can be distinguished by being a simple, multi- or hypergraph, where the latter is a generalization where edges can have sequences of target or source nodes [AEH+99].

Graph transformations are based on applying rules to a graph, while this procedure can be iterated until no rule can be applied any more. These graph replacement rules consist of a left-hand side and a right-hand side. When applying the rule, an occurrence of the left-hand side will be replaced by the right-hand side in the given graph [AEH+99]. In Figure 2.5 a sample graph transformation is shown with one graph replacement rule. Like depicted in the figure, a rule can be rendered by explicitly stating the left-hand side and the right-hand side. Alternatively, the integrated option can be used, where the differences of the two sides are rendered with an additional label and color. An edge or node which only occurs on the right side is labeled ”++” while the label ”--” indicates that they are only occurring on the left side [Win15].

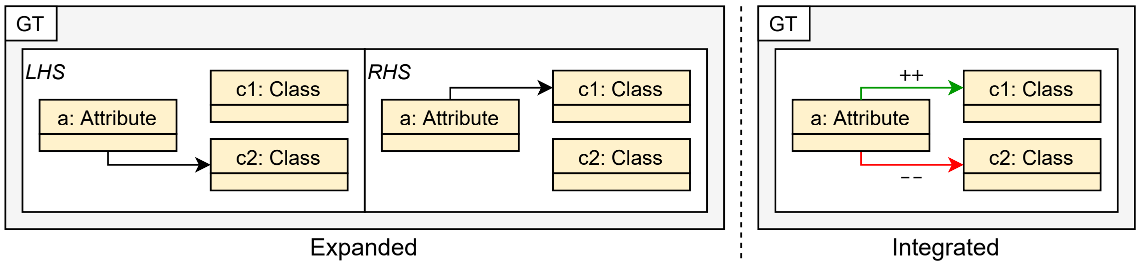

Figure 2.5: Rendering options of graph replacement rules (based on [Win15])

There are three common theoretical approaches of a graph transformation [Win15]:

- Algorithmic: This approach is based on set theory where graphs are described as sets of nodes and edges. Usually nodes and edges can be labeled or attributed to describe models of abstract data types. Complex graph transformations on a given host graph then can be built by using graph pattern matching and graph replacement rules on sub graphs. The procedure of a graph transformation starts with searching the left side of the production in the host graph. Next the nodes of the identified sub-graph are removed. Then the sub-graph of the right side of the production will be inserted and correctly embedded into the host graph. Finally, the attributes of the nodes are set and recalculated if necessary.
- Algebraic: In this approach a generalization of Chomsky grammars is used to define strings as graphs. As a main concept the concatenation of strings is handled as the gluing construction for these graphs. The process of gluing to construct a graph is seen as an ”algebraic construction” which is called *pushout*. As the basic idea general results from algebra and category theory can be applied for these kinds of graphs. Additionally, graph grammars can be used as a generalization of term rewriting systems, which uses trees as terms [Roz97].
- Logical: This is a hybrid approach of the previous two and defines graph replacements with the use of first-order logic. Graph schemes are used to statically describe conditions which are checked as integrity constraints after modifying the graphs [Win15]. One practical implementation of this approach was made in *PROGRES* developed by Schürr [Sch13].

A further relevant approach regarding graph transformations are Triple Graph Grammars (TGG) described by Schürr [Sch94]. TGGs are mainly used for bidirectional model transformations which have a high relevance in applications like model synchronization, round-tripping and realizing editable views [ALS16]. TGGs provide a declarative approach to specify consistency relationships between three different graphs: source, target and correspondence. The correspondence graph is used which primarily allows to synchronize the other source and target graphs. The ideas of a TGG can be used in an algorithmic as well as an algebraic approach [Sch94].

2.3 Tooling and Techniques

This section gives an overview over tools and techniques which are used in this work. First the EMF is introduced which represents the foundation of the tooling. Next, the OCL is briefly described which is used to validate models in the transformation chain of this work. Afterwards, the Xtext framework is introduced, which is used to create the concrete syntax of the DSL for modeling queries of MPWs in Section 5.1.3. In addition to Xtext, the generation of code with Xpand is also illustrated. For modeling commands, the tool Henshin is introduced which allows to visually define in-place graph transformations used as operational behaviors. Finally, QVT-O is described, which plays an important role in this work to perform model-to-model transformations.

2.3.1 Eclipse Modeling Framework

The tooling used in this work is fundamentally based on the EMF which is widely used in industry and research for MDSD. EMF is based on the Eclipse framework and is the core of the *Eclipse Modeling Project* which provides model-based software development technologies. As examples these technologies can be used for model transformation, database integration and graphical editor generation. For this the EMF model can be used to achieve a unification of UML, Java and XML [SBMP08].

The EMF meta-meta-model named *Ecore* is a simplified version of the OMG defined EMOF and like the MOF meta-meta-model specified on the abstraction level M3. Therefore, Ecore is also an EMF model and recursively describes itself. Figure 2.6 shows the Ecore meta-model. The figure is based on the documentation of the EMF version 2.11 [Fou], while it leaves out some redundant details and most derived references to simplify the graphic. Every class shown in the figure derives from the base class `EObject`, which is not shown for simplicity. On the one side `EObject` provides a reflection mechanism where the underlying `EClass` can be obtained by the method `eClass()`. Reflective operations `eGet()` and `eSet()` can be used to modify the state of the object. On the other side a notification mechanism based on the observer pattern is supported. Besides `EObject`, there is another basic class named `EModelElement` which is used for most of the EMF meta-classes to allow that `EAnnotations` can be embedded. `EPackage` objects are used to have a container of `EClassifiers` and they also define an `EFactory`, which is used to instantiate concrete `EObjects` from the types defined in the package. While every type shown in the diagram is usually realized by Java interfaces, with factories the *Abstract Factory pattern* [Gam95] is implemented which decouples the user from the concrete Java classes. To express typed elements in an EMF model the two meta-classes `ETypedElement` and `EClassifier` are used. `ETypedElement` is used for any object which contains a type information like operations, parameters and structural properties. The type information on the one side consists of a reference to a type represented by an `EClassifier` named `eType`. On the other side the type information can be specified in more detail by the attributes `ordered`, `unique`, `lowerBound`, `upperBound`, `many` or `required` which e.g. allows to specify sequences, ordered or unordered sets or single relationships to other objects. EMF classes are represented by the type `EClass` which derives from the meta-class `EClassifier`. The properties `abstract` and `interface` are used to specify if a class is concrete, abstract or an interface. `EClasses` also define a collection of super types by the reference `eSuperTypes`, which allows the use of multi-inheritance. The structure of an `EClass` is defined by the two types `EStructuralFeature` and `EOperation`. With the former structural properties like references or attributes are defined, while the latter allows to define operations with optional parameters. `EStructuralFeature` provides several attributes to be precisely configured, e.g. `changeable`, `unsettable`, `derived` or a `defaultValue`. Further, it provides a feature identifier which can be queried with the operation `getFeatureID()` and used to be identified on the context of the relating `EClass`. While `EAttributes` are relatively simple and can only be typed by `EDataTypes`, with `EReferences` other objects based on `EClasses` can be referenced and the containment relation can be controlled by the `containment` and `container` attributes. Ecore restricts that every containment has only up to one parent container. It is also possible to specify opposite relations with `EReferences`. The type `EDataType` is an alternative to `EClass` to provide a type for an element, which is restricted to primitive types like integers, strings or enumeration literals. Enumerations can be specified by the types `EEnum` and `EEnumLiteral` [SBMP08].

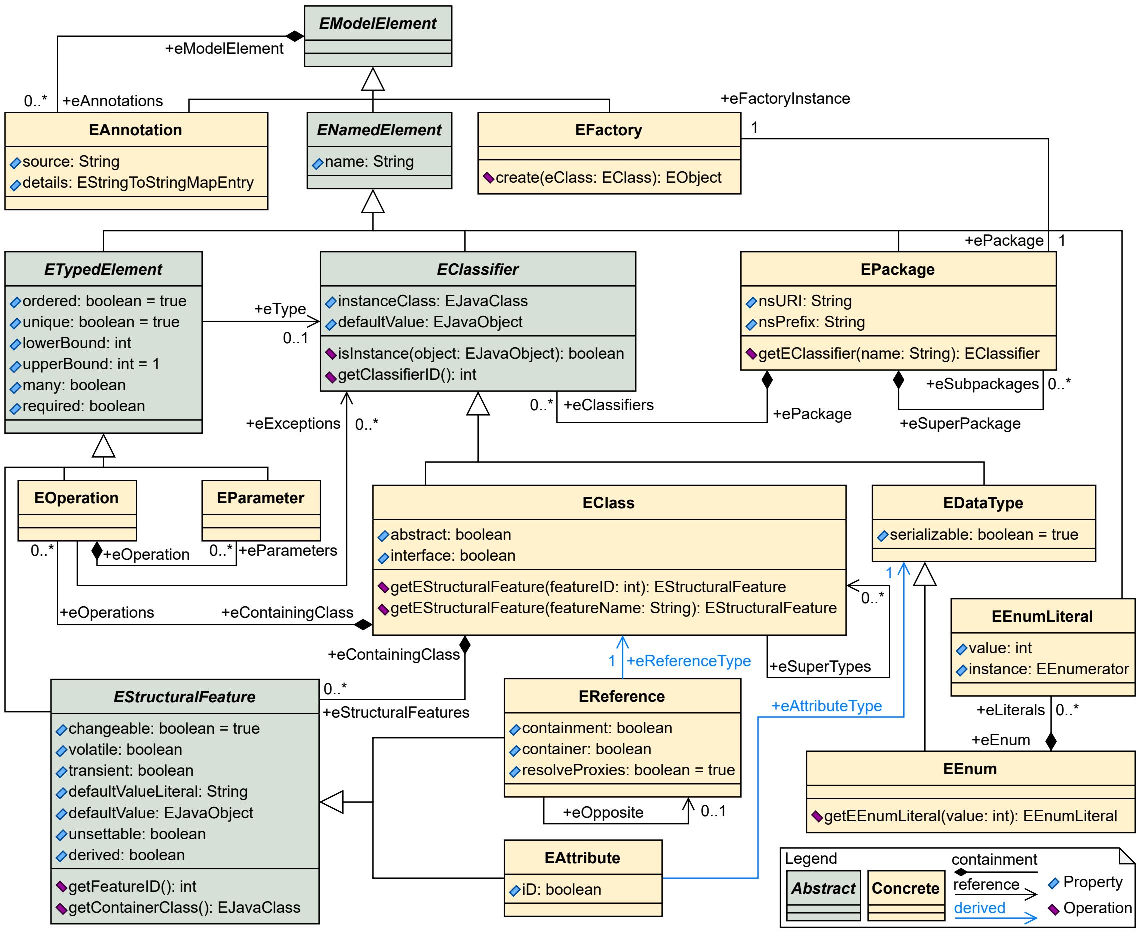

Figure 2.6: Simplified view of the Ecore meta-model (based on [Fou])

XML Metadata Interchange (XMI) is a standard to enable the serialization by XML for modeling and is specified by the OMG [Obj15]. With XMI the exchange of meta-data of models is supported in a standardized way across multiple tools. By default, any instance of an EMF model can be serialized by XMI where the names and the element hierarchy regarding the containment relations are used [SBMP08].

All meta-models defined in this work are based on the Ecore meta-model and make use of most of the details described previously.

2.3.2 Object Constraint Language

OCL is a formal language standardized by the OMG which allows to describe expressions on UML models. These expressions are always side-effect free and used e.g. to specify preconditions, postconditions, invariant conditions or queries. While the expressions themselves are side-effect free, they can specify the behavior of operations altering the state of a system e.g. through postconditions. One main motivation of OCL is to get a possibility to describe constraints on objects in an unambiguous language. It has been developed for being easy to read and write. Since OCL is a typed language, each expression has a dedicated type. An OCL expression hence has to be well formed w.r.t. its type conformance, so used types have to be compatible. Each object which is used from a UML model is typed by its given classifier of the meta-model. Additionally, to custom types, the OCL comes with predefined types [Obj14].

Several modeling tools like Eclipse include OCL for verification and validation features. But often OCL is not included for code generation in modeling tools, instead simplified DSLs are sometimes designed as a workaround [CG12]. This is also the case for specifying constraints and queries in this work. OCL is a helpful orientation, but a dedicated DSL has the advantage of being much simpler and better suited for the needs of this work.

2.3.3 Xtext and Xpand

The Xtext framework is part of the openArchitectureWare project and can be used to build textual DSLs. The input for specifying a DSL with Xtext is written in an Extended Backus-Naur Form (EBNF)-like notation. Xtext then generates Abstract Syntax Tree (AST) classes based on Ecore and a parser which reads the textual concrete syntax of the language. Further, an Eclipse editor will be generated to have syntax highlighting, code completion and static error checking for a given textual syntax [EV06].

When defining DSLs, the base language Xbase can be used which already provides a tight integration into the Java type system. It is used as a base for the language Xtend which is a functional and object-oriented general-purpose language for the Java Virtual Machine (JVM) [EEK+12]. Xtend is integrated into the model-to-text generation template language Xpand which is also part of the openArchitectureWare project. With Xpand textual template files are defined which can import Ecore meta-models and define generation statements based on them. With special characters, dynamic code snippets in templates can be marked based on a given context type. In dynamic code snippets loops, branches or sub-routine calls can be used to implement the behavior of the generator. Using Xtend, it is possible to define extension methods for used meta-types to add behavior or derived properties [EV06].

Xtext is used in this work to specify an input language for queries and constraints, while Xpand is used to generate the final code of the MPW simulators.

2.3.4 Henshin

Henshin is an EMF based tool to define in-place graph transformations, which operate on Ecore models. The underlying approach can be classified as an algebraic approach of graph transformations (see Section 2.2.4 on page 12). Henshin provides a textual syntax for rapid development of transformations. It also supports a graphical editor based on the integrated rendering option depicted in Figure 2.5 [SBG+17]. The modeled graph transformation rules can be translated to the Attributed Graph Grammar System (AGG), which is a tool environment to further analyze algebraic graph transformations [ABJ+10].

The EMF based meta-model for Henshin transformations defines rules consisting of nodes, edges and attributes. Nodes are typed as `EClass`, edges carry the type `EReference` and attributes are modeled by `EAttribute`. Further, rules can define positive or negative application conditions on the level of an attribute, a reference or a node. It is also possible to define graph conditions in a first-order logic, which are formulated in the context of the whole graph.

Henshin defines so-called *units* which are used to invoke and structure the control flow of rules. The simplest unit is a single rule, while more complex units might be used for sequences, conditions or loops. Every unit in Henshin can have multiple parameters, which can be used to provide already bound object variables or simple values to parameterize a unit. To pass parameters from one unit to another, parameter mappings have to be defined which connect a source parameter to a target parameter of the invoked sub-unit [ABJ+10].

The runtime component can be used to execute Henshin transformations. It provides an interpreter engine which allows to define an Ecore model as input and run a selected unit on it. In addition, the runtime component contains a state space generator and an extension point for further analysis [ABJ+10]. Additionally, Henshin transformations can be invoked directly by the Henshin's Java API. This way an engine can be started, which transforms a selected Henshin transformation, based on given Ecore models and parameters [SBG+17].

In this work, the Henshin tool is reused to model commands for target MPW simulators. Since there is no planned code generation feature of Henshin transformations [Win15], a custom code generator for Henshin models will be designed from scratch. With this custom code generator, executable code in target languages like Java and C++ is generated for related Henshin models.

2.3.5 QVT-Operational

QVT is an OMG specification for the three transformation languages *Core*, *Relations* and *Operational*. While the Core and Relations languages provide a declarative approach, the Operational language uses the imperative paradigm [Obj11]. In this work, the QVT-O is used to perform model-to-model transformations, therefore the focus for this section lies on this language part.

In QVT-O it is possible to provide input, output or input/output models for a transformation. The used model types have to be declared in order to be used for a transformation signature. When using QVT-O on the base of EMF, a meta-model can be referenced by its registered URI which points to the Ecore based meta-model. For executing a transformation, matching model instances must therefore be provided to match the right signature. Each QVT-O transformation has to provide a `main` routine which specifies the entry-point and allows to consist of a sequence of QVT-O expressions. Besides `transformation` modules, it is also possible to use `library` modules to achieve a modularization of more complex QVT-O transformations [Nol10].

QVT-O defines three kinds of top-level operations [Nol10]:

- `mapping`: The mainly used operation in QVT-O are mapping operations which operate on input and output model instances and optional further parameters. The general purpose is to map an element of the source model into an element of the target model. If mappings use types of the declared source or target model-types of the transformation, then these objects are automatically bound to the source or target models. By using the keyword `inout`, mappings can also use the source element as the target element. Further, parameters with a modification direction can be provided to control if an object is read-only (`in`), modifiable (`inout`) or as output created by the mapping (`out`).
- `helper`: A helper operation can be used to extract code into a sub-routine which may take parameters. Like mapping operations, the direction of a parameter might be controlled by `in`, `out` or `inout`. Helper operations hence can have side effects on the given objects they operate on.
- `query`: Queries are also operations to extract code into sub-routines. In contrast to helper operations they are not allowed to have any side effects on any object.

Preconditions and invariants can be defined as guards for mappings in addition to the operational body. Preconditions are declared with the keyword `when` and can be used to specify if a mapping shall be executed. Invariants are declared with the keyword `where`. To express predicates or queries, the consequent integration of OCL can be used in QVT-O [Nol10].

3 Related Work

This chapter is dealing with related work which plays an important role as an orientation for realizing the proposed solution. Figure 3.1 gives an overview of the structure of the related work. Basically, three survey questions are used as guidance to cluster the sections in this chapter:

- How to design MPWs?
- How to generate entity models?
- How to generate operational behavior?

The first question is focusing on the design of a MPW which is closely related to previously implemented variants of the hamster simulator and described in detail in Section 3.1 on the next page. The second question deals with the generation of the static part on entity models in multiple programming languages and is part of Section 3.2 on page 23. Finally, the third question deals with the generation of executable code for operational semantics in Section 3.3 on page 24 for commands and in Section 3.4 on page 25 for queries. Further, the two standards Foundational UML (fUML) and Action Language for Foundational UML (ALF) are handled in Section 3.5 as related approaches.

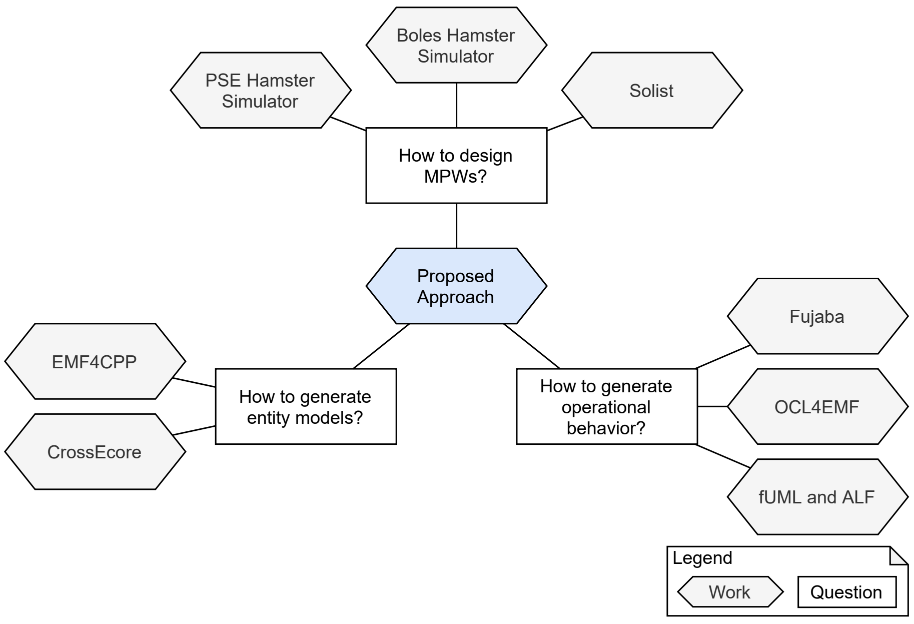

Figure 3.1: Overview of related work

3.1 Mini Programming Worlds

This section describes related implementations of hamster simulators. First, Section 3.1.1 introduces the PSE-Simulator, which is an important reference as the main motivation of this work. Afterwards, the Boles-Simulator as the original hamster simulator and a more generic successor project Solist both developed by D. Boles are described [BB14] [Bol]. Finally, a comparison of the related implementations and the proposed simulator developed in this work is given in Section 3.1.4 on page 22.

3.1.1 PSE-Simulator

For the introductory course PSE at the University of Stuttgart the gamification approach of the hamster simulator is reused. The motivation is to overcome the sheer number of concepts by starting with a simple miniature language. Additionally the didactic is combined with the Outside-In approach described in Section 2.1.1 [BBF20].

The PSE-Simulator[1] is a Java-based re-implementation of the Boles-Simulator described in Section 3.1.2. The re-implementation allows to use modern Java versions and modern IDEs. In sample programs usually a class is derived from a helper class like `SimpleHamsterGame` which hides the `main()` method and prepares an instance of the hamster to be used by a client. The rendering of the simulator is processed by *JavaFX* [2]. When running the simulator, a JavaFX window is popping up which allows that the rendering is processed in parallel to the IDE. To also enable web-based User Interfaces (UIs), the PSE-Simulator has been extended by a client-server capability. This way the core of the simulator can be processed on a server, while the rendering can be separated to be processed on the client side like web-browsers.

Since the PSE-Simulator is based on modern Java and provides a simple API, it is also possible to use other client languages like Scala, Kotlin or even Python. Due to an additional need to support the native languages C/C++, the PSE-Simulator has also been extended by a C-API. However, since the C-API wrapper approach is cumbersome, this thesis strives to achieve the goal that a native simulator core is available.

3.1.2 Boles-Simulator

The scope of this work is oriented towards the hamster simulator by Boles which is a self-contained tool to support the editing, compilation and execution of commands and queries related to a hamster. As a purpose of the tool, Boles mentions that the simulator shall provide a simple didactic way to teach programming and let the focus of the learner be on the problem-solving aspects and not on technical tooling issues. The simulator world consists of a two-dimensional, grid-based territory of tiles where one or more hamsters, grains or walls can be placed. For a hamster, the four base commands `vor()`, `linksUm()`, `gib()` and `nimm()` are available to move the hamster, turn the hamster

[1] https://git.rss.iste.uni-stuttgart.de/open-to-public/pse

[2] https://openjfx.io/

to the left, put a grain on the hamster's current tile or let a grain pick one from it. Furthermore, the three queries `vornFrei()`, `maulLeer()` and `kornDa()` are supported to check if the front is clear, if the mouth is empty or if any grain is available on the hamster's tile.

The self-contained tool developed by Boles has its own debugger and instruments the written Java text to be compiled with the original Java compiler [3]. Its integrated editor has the drawback, that it has no auto-completion features to assist the learning programmer with code suggestions and no simple navigation to other methods is possible. The integrated debugger is not so powerful as modern debuggers, e.g. since it does not allow to set breakpoints or does not provide advanced features like mutation of variables. Compared to modern IDEs, there is no built-in support for a versioning control system given [BB14].

One notable advantage of Boles' hamster simulator is that it allows to use multiple programming languages. Even visual ones are supported, like Scratch, finite state machines, program-control-flows or structograms. While programming languages like Scheme, Prolog, JavaScript, Python and Ruby are can be used, no support for C++ is given, which is one of the main goals of this work [HamsterModell20]. As a further distinction, the simulator created in this work shall run in modern IDEs with full refactoring and code assistance support.

3.1.3 Solist

A successor project to the hamster simulator is the Solist tool. This has also been developed by Boles and it allows to build other simulators for a MPW as well. Popular examples besides the hamster simulator are *Kara the ladybug* or *Turtle-Graphics*. While Solist represents a generic approach where custom MPWs can be created, they are more lightweight as the original hamster simulator. For example, they are currently only available for the language Java and the visual language Scratch. As a basis of the approach a meta-model is defined which is oriented on a theater and depicted in Figure 3.2. Important classes in the meta-model are `Stage`, `Component`, `Actor`, `Prop` or `Performance`. As a further restriction in the Solist tool only one actor can be created, hence it is not possible to place more than one hamster on the territory [Bol].

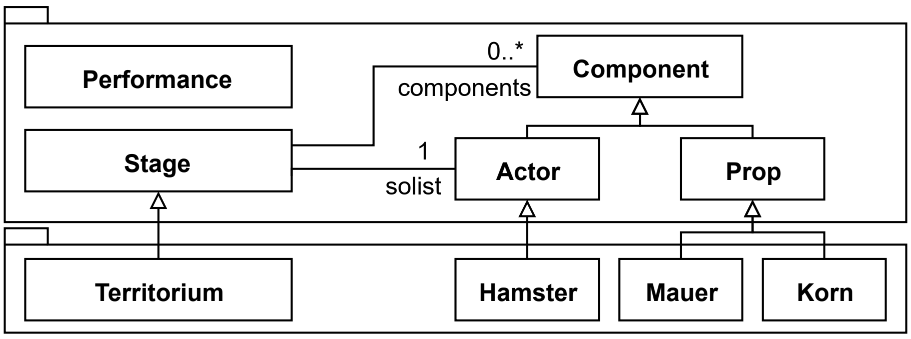

Figure 3.2: Excerpt of the meta-model of Solist [Bol]

With its already established meta-model the project Solist is an important related work for this thesis. It gives a helpful orientation to create a modeling approach which also supports other MPWs.

[3] Like mentioned in the manual, the Java `tools.jar` is included in the simulator

3.1.4 Comparison of Mini-Programming-World Simulators

Table 3.1 summarizes the different aspects of the Boles-Simulator, Solist, PSE-Simulator and Proposed-Simulator.

Aspect	**Boles-Simulator**	**Solist**	**PSE-Simulator**	**Proposed-Simulator**
Didactics Approach	algorithm-first	algorithm-first	object-first (Outside-In)	object-first (Outside-In)
Native Platform	Java	Java	Java	Java, C++ (*)
Client-IDE	integrated	integrated	external	external
Client-Language (Code)	Java, Python, Ruby, JavaScript, Prolog, Scheme	Java, Scratch	Java, Python, C++	Java, C++ (*)
Client-Language (Visual)	Control-Flow-Diagram, Scratch, Automata, Structograms	Scratch	-	-
Remote Client	no	no	yes	no
Multi-Language Approach	interpreter & compiler	interpreter & compiler	API-wrapper	generated code
Mini-World Adaptability	no	yes	no	yes

(*) more languages might be extended in future

Table 3.1: Hamster simulator aspects

First, each *didactics approach* is highlighted where both the PSE-Simulator and Proposed-Simulator are designed for courses based on Outside-In. For example, this is notable by the principles of Design of Contract which is given by the hamster's API.

The second aspect looks at the *native platform* which states on which platforms or programming languages a simulator's core is natively available. While the core of the Proposed-Simulator is generated in multiple programming languages like Java or C++, the other ones are written in Java.

Next, the usage of the *client-IDEs* is compared. Both, Boles-Simulator and Solist are built as a standalone tool with their own integrated editor. In contrast, the other both are designed to be developed in an external IDE like IntelliJ or Eclipse. For this, they are using a build management tool like Maven. While an integrated IDE simplifies the setup of the simulator, the external one comes with more powerful and modern coding assistance and debugging support.

A further aspect is the supported set of languages which can be used by a client. The Boles-Simulator supports several coding-languages and also several visual ones. Simulators built with Solist are simpler variants of the Boles-Simulator and support fewer languages. In contrast, the PSE-Simulator and Proposed-Simulator support the native language C++. Visual ones are not available for them.

The implementation of a *remote* feature is only done by the PSE-Simulator. It allows to distribute the client side to other machines and also enables the usage of web-IDEs. This avoids that students have to setup an IDE by themselves. While the Proposed-Simulator is currently not supporting any remote features, it follows the *Humble Object pattern* [Fow20] to provide a simple view model for rendering. This approach simplifies the extension of a remote capability in the future.

Since every approach provides more than one client language, additionally the *multi-language* approach is compared. Boles-Simulator and Solist are both using internal compilers or make use of interpreters to execute non-Java code. The PSE-Simulator makes use of an API-wrapper. For example, to allow that clients can use C/C++, a native C-wrapper is used to internally start a JVM and control it via this API. In contrast, the Proposed-Simulator uses a code generation approach where the simulator with its core are completely available in the client languages like Java or C++. This avoids third party tools to integrate the client languages, simplifies error handling and allows clients do dig into the internals of the simulator's core.

Finally, the *mini-world adaptability* is noted. Only Solist and the Proposed-Simulator are designed to create multiple MPWs. While Solist allows to do this in the Solist tool, for the Proposed-Simulator a full setup of the modeling environment is required.

3.2 Model Generation

In this section, projects related to the generation of EMF models into code for multiple programming languages are compared. As one important requirement of this work especially the code generation to the languages Java and C++ shall be possible, hence approaches are being researched to generate EMF models in further languages besides Java.

With *EMF4CPP*, an EMF-like code generator for C++ has been developed. The motivation has been the weak support of other programming languages besides Java, especially C++ [JMJ+16]. The generator is implemented with Xpand and Xtext and C++ 11 libraries are generated as output [EMF4CPP20]. Many EMF features are supported like reflection, bi-directional relations and an EMF conform meta-model. Jäger et al. mention that special challenges are to implement the multi-inheritance relations of EMF, the reflection capabilities and a consistent memory management. As a limitation, currently no OCL generation is supported [JMJ+16].

Another approach which targets to adapt EMF to other programming languages is *CrossEcore*. It is a multi-language approach which allows to generate C#, Swift, TypeScript and JavaScript from Ecore models with embedded OCL expressions. While many code generators exist, which generate code from Ecore models into languages like Java, C#, C++ and more, only for the languages Java and C# the generation of OCL is also provided. As a solution, CrossEcore provides a multi-platform modeling framework which allows to be extended for new programming languages. But no generation for C++ is supported at the moment [SJGE18].

In this work a model generator is written for Java and C++. While the combination of CrossEcore and EMF4CPP would allow to generate models in several languages including C++, especially OCL expressions are currently not available for C++. A generator which makes use of the simplicity of the MPW domain can circumvent the gap of missing support of generation features for a programming language. In addition, several other advantages can be identified. On the one side this allows that the generated code is similar in both programming languages. On the other side it allows much more flexibility, e.g. it allows to define stereotypes for value-types which are generated as simple data-structures and not as an Entity class. Another main motivation is that the generated code shall be used to teach programming, so the focus of the generated code is also on readability and not only on performance aspects. As an example, for C++ it is helpful to use smart pointers like `std::shared_ptr` as defined in the standard.

3.3 Generation of Graph Transformations with Fujaba

As a further important related work, the tool environment Fujaba is described in this section which allows to generate executable code from graph transformation rules. Fujaba is a Computer Aided Software Engineering (CASE) tool which integrates UML and Java. This allows the development of applications by the usage of graph transformations [GZ06]. It is categorized as an algorithmic approach of graph transformations [Win15]. Fujaba stands for *From UML to Java and back again* and relies on the Story Driven Modeling (SDM) approach, which has been prominently realized by Zündorf in his work *Rigorous Object Oriented Software Development*. With this approach, Zündorf tried to fill the gap between high-level aspects like use-cases and the formal description of the aspects of a software system [Zün01]. The main concepts of Fujaba is to use so-called *Story Diagrams* to describe behavior, which are a combination of UML like activity diagrams and graph rewrite rules [NNZ00].

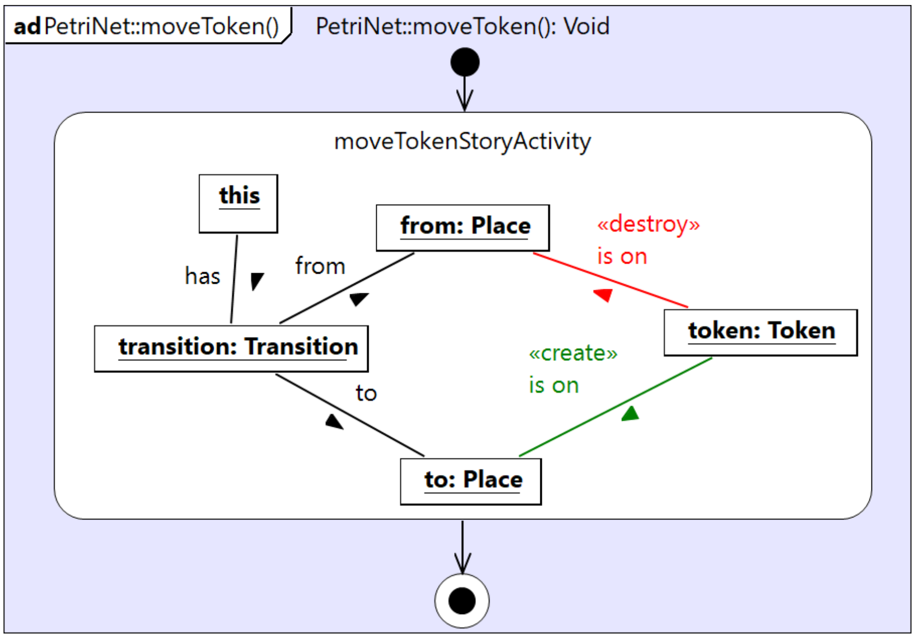

Figure 3.3: Fujaba Petrinet sample

In Figure 3.3 an example of such a Story Diagram is shown. Each diagram has a start and one to many end nodes similar to UML activity diagrams. A graph rewriting rule is placed as a special activity, which looks like a UML object diagram and represents graph patterns. These graph patterns have to be matched for executing the rule. In the example there has to be an object of type `PetriNet` which is represented by the `this` object. There must also be one object of type `Transition`, two objects of type `Place` and one of type `Token`. The red link marked with `<<destroy>>` will be deleted when the activity is executed, and the green link marked with `<<create>>` will be created.

Another main concept of Fujaba is that it has a round-trip capability. This means that written Java code can be used to reconstruct Story Diagrams. This feature allows that the generated code can be adjusted for testing and maintenance reasons on-the-fly, e.g. for global-search-and-replace refactorings [NNZ00].

Fujaba has successor projects and is no longer actively maintained. Zündorf mentioned that the motivation has been GUI problems. The direct successor of Fujaba is the *SDMlib*, which is also Java-based but allows to express the operational behavior by *tables* in Java code. In this approach it is also possible to parse the written code and then let the visual representation like UML object diagrams be generated. In 2018 the SDMlib was re-implemented by the project *Fulib* which stands for *Fujaba library* [Zün19].

Fujaba is developed for Java, since the main goal was that it is based on a well-accepted object-oriented programming language. C++ has been also a candidate for the development, but due to its complexity compared to Java it has not been chosen [Zün01]. While Fujaba is a tool to develop any kind of system, in this work the target systems are MPWs and hence many assumptions can be made to overcome the general complexity of C++ as a target language. Nevertheless, the ideas how Fujaba solved the generation of Java code from visual graph transformation patterns are used as an important orientation in this work.

3.4 Generating OCL to Code

Besides the compilation of operational behavior by graph transformations which is described in the previous section, in this work another aspect is to compile queries and constraints into executable code. During the development of the project *OCL Compiler for EMF*, the authors Garcia and Shidqie stated that the tooling to compile models and also their model constraints together into code were often not provided. This motivated them to fill this gap for the EMF by developing an integrated OCL compiler to translate OCL to Java. They focused on an Eclipse plugin for their work, to allow them to generate code from OCL expressions attached as annotations in Ecore models. Their approach is to hook into the *GenModel* mechanism of the EMF code generation and let the OCL expressions be parsed, which are contained in `EAnnotations` attached to the model elements. For designing the architecture, they identify two approaches to generate expressions into code. The first approach is to perform a direct translation from an OCL AST into Java code. As a second approach, one can define an intermediary model, which contains imperative statements. These imperative statements can be more easily generated into Java expressions. This second approach is also used by their project [GS07] [SMGG07].

AOCL is a more recent project to compile OCL into Java code which uses Java streams and lambda expressions. It provides a similar language which relies on relational algebra and allows a simple generation of executable code. The motivation is that OCL is often criticised to be too complex and difficult to learn. Another motivation is that previous projects like the *OCL Compiler for EMF* have been completed before Java 8 and hence no use of lambda functions is made, which leads to a more verbose code [BA20].

Regarding the discussed related projects, in this work a simple query language oriented on OCL is designed using Xtext. As mentioned before, the previous projects are often based on OCL which brings its own complexity. This is not needed for the relatively simple constraints and queries of a MPW. Besides that, an approach is intended which allows that the generation of code shall be easily adaptable to other programming languages like C++. While approaches like *CrossEcore* already support the generation of Ecore and constraints into multiple programming languages, especially the language C++ is missing and there is currently no other multi-platform approach which fills this gap. Furthermore, the custom language has the advantage that it can be adjusted for graph transformations and hence can also be reused e.g. for attribute conditions. To have a helpful orientation for the generation of the MPW queries and constraints, the related projects like *OCL Compiler for EMF* are used as a reference how the generation of OCL expressions can look like.

3.5 Executable Modeling with fUML and ALF

The OMG defines two standards fUML and ALF for modeling of executable semantics in the context of UML. With fUML, a subset of the UML is defined, which is computationally complete and provides precise execution semantics. Based on fUML, the ALF defines a standard of a textual concrete syntax to represent fUML models [Obj13a][Obj13b].

Schröpfer and Buchmann developed an Eclipse based approach to integrate ALF and static modeling aspects by the UML. This enables a fully executable Java code generation approach. They reused a graphical UML editor for modeling structural aspects, while they developed a concrete ALF syntax with Xtext for behavior modeling. To enable a consistent integration, the static model for ALF and the visually modeled abstract syntax are synchronized with a bidirectional model transformation approach named *BXtend* [SB19].

Guermazi et. al describe experiences with fUML and ALF based on the open source tool Papyrus[4]. They focused in their paper on the concerns extensibility, control and observability, time support and connectivity. Most identified limitations can be improved by further enhancements of the tooling support [GTC+15].

Bedini et. al. further developed a fUML execution engine for C++, which allows to execute models based on given class and activity diagrams. With *fUML2C++* and *UML2C++*, two generators are implemented which takes a fUML model and generates C++ source code. Linked with further provided libraries, an executable software is realized [BMW+17].

The standards are addressing similar problems, which have to be solved by this work. For example, the graphical modeling of control flows by activity diagrams are comparable to control flow units in Henshin. Henshin also provides a textual concrete syntax, but in contrast to ALF, the focus lies on the declarative modeling of rule transformations.

[4]https://www.eclipse.org/papyrus/

4 Concept

This chapter illustrates the main concepts used to develop the proposed solution. First, in Section 4.1 a brief overview of the most important requirements is given. Then, an introduction for the central MDSD approach is depicted in Section 4.2. Afterwards, Section 4.3 will outline the module structure designed for the MPW modeling framework, which includes most of the relevant artifacts related to the MDSD approach. Next, Section 4.4 will move the focus on the simulator architecture, which defines the basic building blocks of the concrete MPW simulators. Based on this, Section 4.5 will show the design of the central simulator core model in more detail, including its client API, commands and relevant meta-models. The ongoing sections will focus on further non-functional aspects. Section 4.6 is about the testing strategy used in different layers and development phases. Then, Section 4.7 briefly states how different approaches for documentation in the final MPW simulators are applied. Next, in Section 4.8 the adaptability of the proposed solution is described. On the one hand, the adaption to new programming languages is illustrated, while on the other hand the modeling of further MPWs is outlined. Finally, Section 4.9 will give a complete overview of the used tools and technologies.

4.1 Requirements

This section gives an overview over gathering and specification of requirements for implementing this project. Most functional requirements are derived directly from analyzing the existing solutions. Therefore, the first phase in this project has been to analyze and specify the requirements in a structural manner.

To give an overview, some main goals of this project are as follows:

- Commands of the hamster simulator shall be modeled by Henshin.
- Queries and constraints like pre- and postconditions shall be modeled with an OCL-like syntax.
- Entity models and operational behavior shall be generated for at least the two languages Java and C++.
- Other MPWs like *Kara the ladybug* shall be possible to be modeled as well.
- The core API shall be similar to the API of the PSE-Simulator.

Other related works like the *PSE-Simulator*, *Boles-Simulator* and *Solist*, as illustrated in Section 3.1.4, already provide working implementations for the hamster simulator. To provide a C++ implementation is one main requirement, which is not fulfilled by these other works. Further, they achieve a multi-language approach based on interpreters, internal compilers or API wrappers, where the simulator's core implementation is not available in the target language. This shall be realized in a different way, e.g. by generating the simulator's core for the target language. In this way, it will be easier for students to dive into internals without being confronted with technical boundaries. Finally, the concepts of Design by Contract shall be supported in a formalized way. The PSE-Simulator deals with Design by Contract by specifying constraints as Java Modeling Language (JML) statements. But relating statements in executable code have to be manually added, therefore it only provides a formalized documentation. With the approach in this project, pre- and postconditions shall be supported as first-class aspects, which allow that relating code can be generated.

While the previously mentioned requirements are the most relevant ones, the full list can be found in the wiki of the code repository[1]. To be able to classify these requirements, following categories are defined:

API: Specify requirements for the final simulator API, which is used by clients like students.
PLT: Specify platform requirements of the final simulator like the portability to Java and C++.
UI: Specify UI simulation requirements like graphical representation of the game state.
HCMD: Specify concrete hamster commands to control the hamster in the simulation world.
TCMD: Specify concrete territory commands to build the territory and its tile contents.
HQRY: Specify concrete hamster queries, to query information about the simulation state.
MDE: Specify requirements for the MDSD approach to develop the simulator.
NONF: Specify non-functional requirements.

Each concrete requirement is specified in a tabular structure which is shown in Table 4.1. Every requirement has a unique identifier, which consists of the category identifier and a number. Next, a short summary of the requirement is given. Additionally, a user story describes the requirement in more detail from a user's perspective. Finally, a priority is specified, which is used as an important orientation for time planning. The priority *MUST HAVE* defines the highest one, which has to be implemented by the thesis. *SHOULD HAVE* is planned to be implemented, but on time problems these requirements might be omitted. The lowest priority is marked by *NICE TO HAVE*, which indicates that a requirement is not necessarily to be implemented by this thesis.

ID	Summary	User Story	Priority
HCMD-010	Hamster-Model provides a Move-Command	As a student I want to move the hamster to the front tile.	MUST HAVE

Table 4.1: Example requirement

[1]https://github.com/SQAHamster/mpw-modeling-framework/wiki/Requirements

4.2 Model-driven Software Development Approach

MDSD is used as the basic approach to develop the MPW simulators in this project. Besides the concrete simulators, like the hamster simulator, it covers also a framework part. Further, it can be separated in the modeling and implementation of the simulators. This results in the four segments *MPW modeling framework*, *concrete MPW modeling*, *MPW simulator framework* and *concrete MPW simulator* to be distinguished in the development.

Figure 4.1 gives an overview of these four segments, showing the two dimensions *modeling* vs. *simulator* and *MPW framework* vs. *concrete MPW*. The modeling workflow and the simulators are each developed in different development environments. Hence, in the following the term *modeling environment* relates to the modeling segments, while *simulator environment* covers the simulator framework and concrete simulator parts.

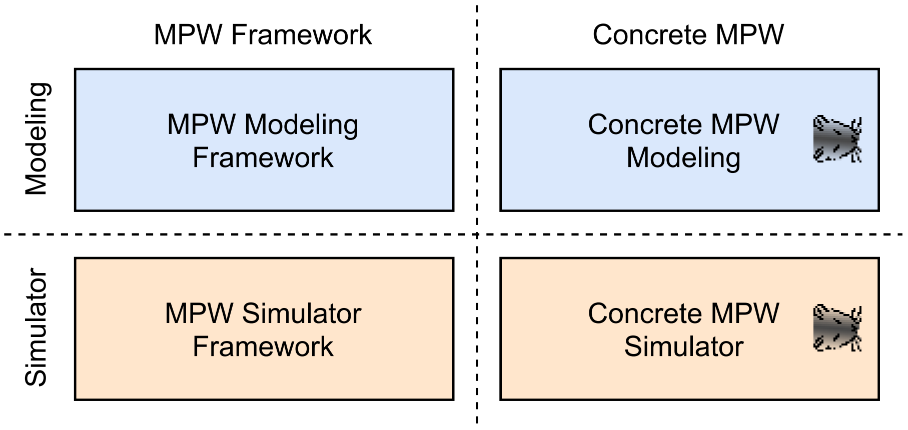

Figure 4.1: Four segments of the MDSD approach for developing MPWs

On the upper left, the *MPW modeling framework* is placed, which is developed in the modeling environment. It contains the MPW meta-models described in Section 4.5.2, Section 4.5.3 and Section 4.5.5. Further, it covers main parts of the modeling workflow. The modeling workflow is key of Chapter 5 and consists of three phases for input modeling, model-to-model transformations and code generation. The modeling framework is published as an OSGi bundle to allow reusing by separate Eclipse environments. On the upper right side, the *concrete MPW modeling* is shown. This segment is based on the MPW modeling framework and includes the input modeling for a concrete MPW like the hamster simulator's entities, commands and queries. Input modeling for concrete MPWs is described in Section 5.1.

On the lower left side of Figure 4.1 the *MPW simulator framework* is depicted. It is part of the simulator environment and represents the basic framework for a concrete simulator. The meta-models of the modeling framework are generated into concrete code like Java. Further classes are implemented, which are reused for concrete simulators. Examples are the implementation of the game control, primitive commands and further utility classes. The simulator framework is published as a library for each specific target programming language. Last, at the lower right side, the figure shows the segment for the *concrete MPW simulator*, which is the result of the concrete MPW modeling segment. It depends on the simulator framework by including the related library and contains the final implementation for a concrete MPW simulator. In case of the Java based hamster simulator, it provides a JavaFX application to visually show the execution of a hamster simulator program.

4.3 MPW Modeling Framework Modules

This section gives an overview of the modularization for the MPW modeling framework. The modules are implemented in Eclipse using the EMF framework and are structured to be deployed as an OSGi component. The structure is oriented on the best-practice layout for publishing with Tycho[2] by using sub-folders named *bundles*, *features* and *releng*[3]. Figure 4.2 shows a package diagram representing the relevant modules of the MPW framework. While the related Eclipse module names have the prefix ”de.unistuttgart.iste.sqa.mpw”, for readability this is omitted in the figure.

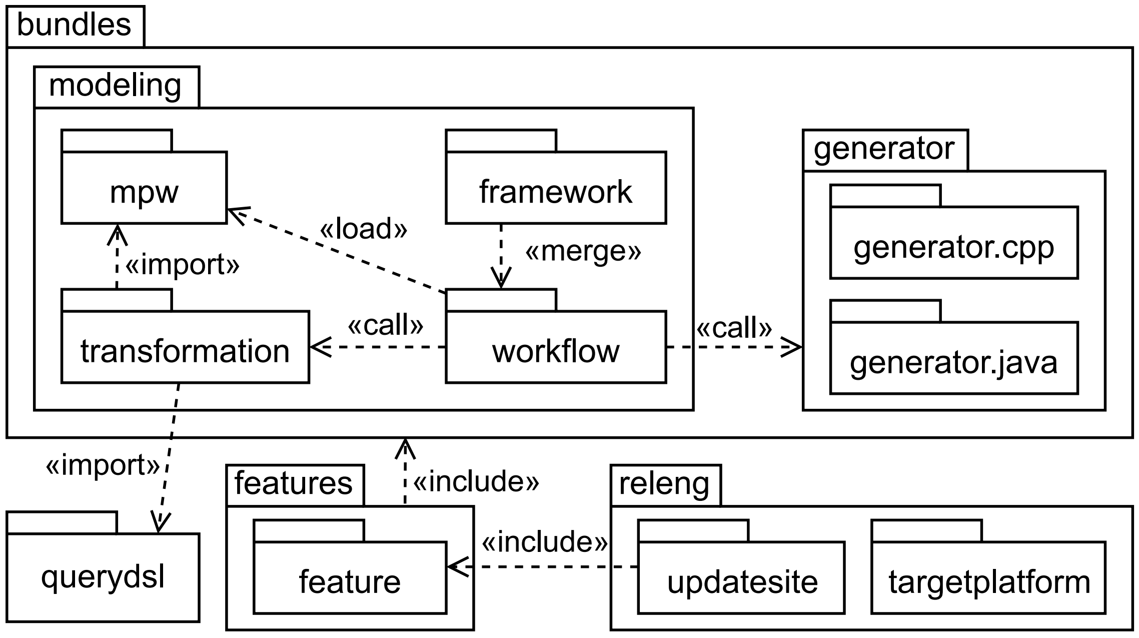

Figure 4.2: Modules of the MPW modeling framework

The modules placed under *bundles* are Eclipse plugin projects which represent the essence of the MPW modeling framework. These plugins are classified as *modeling* or *generator* modules, while the former contain entity models, transformations and workflow scripts. First, the module `mpw` contains meta-models used to describe MPWs and contains the *MiniProgrammingWorld package* (Section 4.5.3), *Command package* (Section 4.5.2) and *ViewModel package* (Section 4.5.5). Transformation logic is placed in the `transformation` module, which includes QVT-O transformations and further intermediate meta-models. It imports `mpw`, since some entity types are transformed for the simulator, e.g. by adding roles to concrete `Actor` or `Stage` classes. Section 5.2 gives more information about the contents of this module. Next, the `workflow` module provides the basic infrastructure to

[2]see https://sdqweb.ipd.kit.edu/wiki/Maven_Tycho
[3]release engineering

execute the transformation workflow based on the Modeling Workflow Engine 2 (MWE2). It defines several classes to load different input models on the file-system, integrates the QVT-O Java API to execute transformation rules and calls the generator components. Since the `workflow` module is not intended to be executed directly, a separate `framework` module is defined. This module contains a concrete MWE2 workflow to generate MPW simulator framework classes. For generation, primarily the two modules `generator.cpp` and `generator.java` are used. They contain Xpand and Xtend code to generate the intermediate models after executing transformation rules into executable code. To share common generation logic, a further module `generator.utils` is defined, which is omitted in Figure 4.2 for simplicity. Section 5.3 will provide more information about these generation modules. Another relevant component is represented by the `querydsl` module (Section 5.1.3), which defines the concrete and abstract syntax of the Query-DSL, which is used to model queries and constraints. It is deployed as a separate OSGi bundle with its own modularization structure. The `transformation` module imports `querydsl` to operate on these input models. The next modules are mainly used for deployment of the plugins projects. While the `feature` module represents an OSGi feature including the *modeling* and *generator* plugin projects, the `updatesite` defines a repository consisting of this feature. Last, the `targetplatform` module defines several update sites, which provide all required OSGi dependencies used in the MPW modeling framework.

4.4 Simulator Architecture

In this section, the target architecture is depicted, which is used for the MPW simulators and covers the *simulator environment* introduced in Section 4.2. A schematic view on the architecture is shown in Figure 4.3. Basically it defines a layered architecture which uses the *Humble Object pattern* [Fow20] to decouple any UI framework from application logic. Additionally it separates UI logic from the simulator's core.

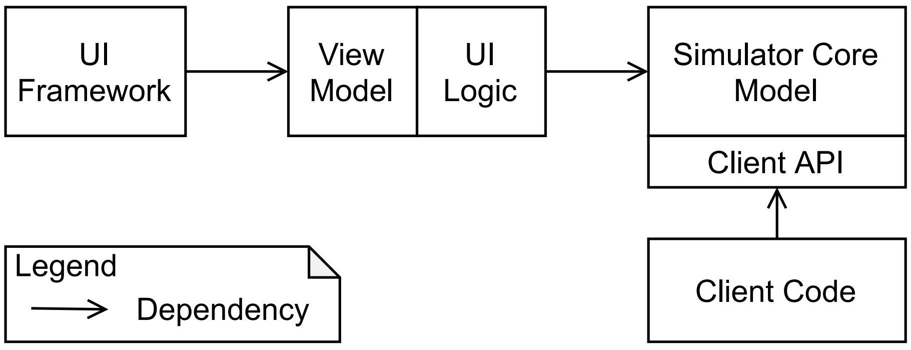

Figure 4.3: Simulator architecture

The *Simulator Core Model* represents the main component, where most of code is generated and modeled behavior is integrated. It provides an API for clients to create games, build stages or interact with actors by their commands and queries. For internal access, the core model provides several *roles* which represent dedicated interfaces for use-cases like reading information, processing game commands or building a stage. Further, state information can be observed by the observer pattern [Gam95]. More details of the core model are described in Section 4.5.

The *UI Logic* and *View Model* represent another combined component, which depends on the core model. By using a view model, relevant information which is rendered on screen is defined as an in-memory representation. It acts as a simple, observable data-structure which can be observed by code based on a concrete UI framework. A presenter class represents the implementation of the UI logic and is responsible to fill or update the view model. For the UI component, only data-structures and class stubs are generated. The UI logic itself has to be implemented manually, since generating this code is out of scope of this thesis.

On the left side the *UI Framework* component is depicted, which is handled as an infrastructural detail of the architecture and therefore placed on an ”outside” layer [Mar18]. It is responsible to render the view model on the screen and obtain user input events to trigger the UI logic. In case of Java, the framework JavaFX[4] is used. For the C++ variant, the Simple DirectMedia Library (SDL)[5] is used as an application framework. Alternatively, this component can be replaced by a test runner, which can interact with the whole UI logic for automated testing.

Last, the *Client Code* is shown below the core model. It represents code which is implemented via an IDE of choice, like Eclipse[6] or IntelliJ IDEA[7]. Called commands on the core are processed with a delay to be able to follow changes on the rendered view.

Since the main focus of this thesis lies on the modeling and generation of the core model and view model, in the following section the design of these components are described in more detail.

[4]https://openjfx.io/
[5]https://www.libsdl.org/
[6]https://www.eclipse.org/
[7]https://www.jetbrains.com/idea/

4.5 Mini-Programming-World Core Design

This section is about the core design in concrete MPW simulators. First, the API exposed to clients by a facade is depicted. Then, the design of commands for undo and redo functionality is illustrated. Afterwards, the central meta-meta-model named *MiniProgrammingWorld* is explained, which covers central meta-types used to define MPWs. Additionally, the idea to design different role interfaces for the reading of information or game and editor use-cases is described. Finally, the view model design is also highlighted as an important concept.

4.5.1 Facade

To provide a convenient API such that clients can easily perform game commands and queries, the core design makes use of the *Facade pattern* [Gam95]. Figure 4.4 gives an overview of the main components which make up the facade for a MPW like the hamster simulator. Basically, the facade API provides several operations to support the main use-cases, like to control the game state, perform game commands, gathering information by queries or building up a stage by editor commands. While the design is independent of a concrete MPW, in following it is described by the example of the hamster simulator.

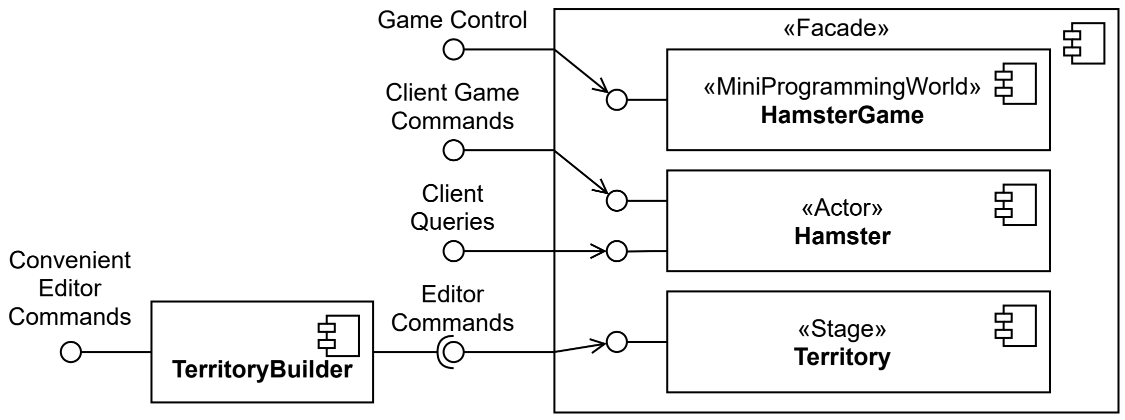

Figure 4.4: Facade of the core model

The `HamsterGame` represents the root class which derives from the meta-class `MiniProgrammingWorld`. On the one side, it provides several operations to control the game state, like starting the game, pausing the game or resuming the game. It also provides methods to undo or redo commands. On the other side it acts as the container of the stage of a MPW.

For clients, the `Hamster` class is usually the most important one, which inherits from the meta-class `Actor`. It provides *game commands* to let clients interact with the MPW in a simplified way. Examples are moving a hamster or let it pick or put grains for its current tile location. Besides game commands, it also provides queries. They represent operations to allow to gather dedicated pieces of state information. Examples relating the hamster simulator are to query if a wall is in front of the hamster or if a hamster has grain in its mouth.

The third component is the `Territory`, which derives from the meta-class `Stage`. It contains tiles, actors and props which represent the main contents of the MPW. Clients usually do not interact with a stage directly, but indirectly with the actor commands and queries. For building up contents of a stage, dedicated *editor commands* are supported. By making use of the `TerritoryBuilder`, any editor command can be invoked by a fluent API, which is implemented by the *Builder pattern* [Gam95]. To load pre-defined territories, clients can make use of the class `TerritoryLoader` which parses serialized territories from an input stream, e.g. based on a resource file. This class internally uses the builder API and hides the editor commands completely from clients.

The previously described classes, which make up the facade, are generated by a code generator. They internally delegate invoked operations to concrete implementation classes, which implement the operational behavior of a MPW. These internal classes are described in the following subsections in more detail.

4.5.2 Commands Package

In this subsection, the *Command package* is described. It is based on the *Command pattern* [Gam95] and defines generic base classes, which are used to encapsulate certain behavior represented by an object.

Basically, a command provides *execute*, *undo* or *redo* of behavior which can be triggered by a client. The concrete implementation of commands supporting to perform undo and redo functionality can be done in one of following two variants:

1. *Complex Commands*: In this variant, one class is implementing the behavior for executing or reverting operational behavior of a command like moving the hamster. As a contradiction to composite commands, they do not make use of primitive child commands. In addition to the ”forward” logic, the ”backward” logic is also implemented explicitly in the complex command class. One advantage of this variant is a simpler class design, since no composite or primitive commands are necessary. As a disadvantage, the code generation of such commands is more difficult, since the reverse case has to be explicitly regarded. Another disadvantage is, that destructive actions like clearing the territory have to notice the previous state e.g. as a full copy of the territory.
2. *Composite and Primitive Commands*: In contrast to complex commands, this variant makes use of primitive commands. These commands are defined in a central place and might be manually implemented. They represent primitive modifications like modifying a property, adding an entity to a collection or removing an entity from a collection. A command like moving the hamster is represented by a composite command, which defines a sequence of primitive commands. The main advantage of this variant is, that the composite-undo is simply the undo of each primitive command in reverse order. This makes generation of code much simpler.

To simplify generation of code in this project, the second variant is used. In Figure 4.5 an Ecore class diagram is shown, which is used for code generation described in Section 5.3.1.

An interface `Command` defines the abstract base type of any command and provides the three abstract methods `execute()`, `undo()` and `redo()`. One implementation of this interface is the abstract class `CompositeCommand`, which makes use of the *Composite pattern* [Gam95]. Each concrete command

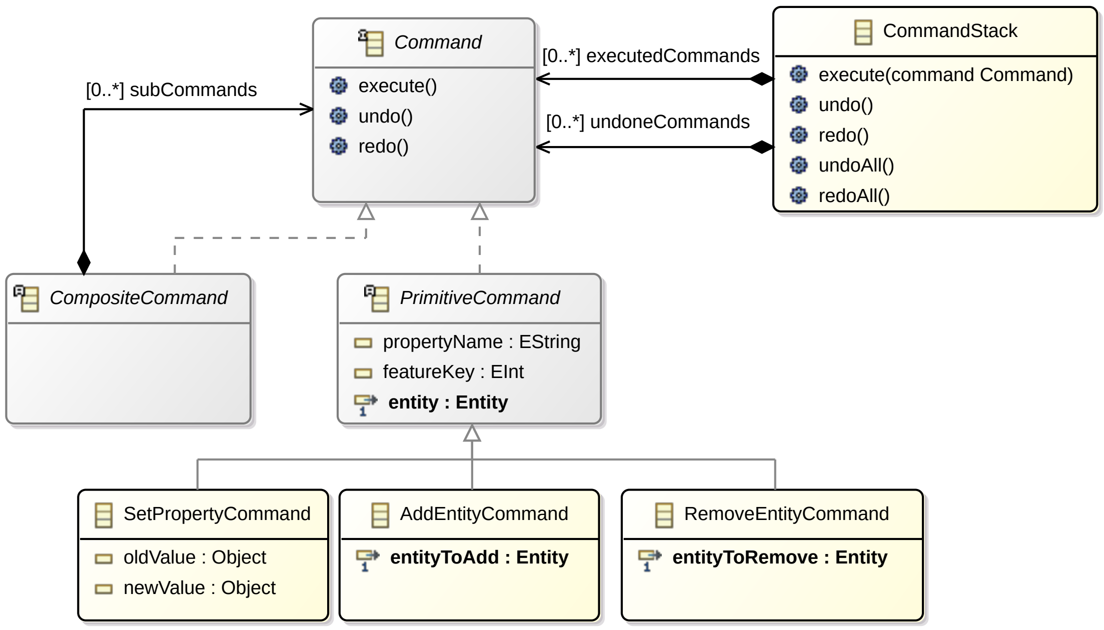

Figure 4.5: Command package

like moving the hamster is generated as a subclass of `CompositeCommand`. The other implementation of the `Command` interface is the abstract class `PrimitiveCommand`. Since each primitive command is based on a concrete property of a target entity, it defines a `propertyName` and `featureKey` which are used to uniquely identify a structural feature of the target entity's class. Additionally, the target entity is noticed in the field `entity`, on which the primitive command performs the modification. There are three sub-classes `SetPropertyCommand`, `AddEntityCommand` and `RemoveEntityCommand`. While the first one is used to modify a single value of the given property, the latter two implementations deal with collection properties. For undoing a `SetPropertyCommand`, the value at time of execution is stored in the field `oldValue`, while the new value is stored in `newValue`. These two fields are typed with `Object`, which indicates that they take any value. In Java it maps to the base class `Object`, while in C++ a custom type `Any` is implemented as a union. An undo of the execution of `AddEntityCommand` or `RemoveEntityCommand` is simply the reverse action, like removing the entity which was added by an `AddEntityCommand` instance.

To store the list of executed commands, a separate class named `CommandStack` is modeled. It keeps track of executed and undone commands to let a caller undo and redo any command in the right order. In contrast to the design of the PSE-Simulator, a command stack has no responsibility to control the mode of the game. This is implemented by a separate class named `GamePerformance` which is described in Section 4.5.3.

4.5.3 MiniProgrammingWorld Package

The internal structure of a simulator's core is mostly represented by the *MiniProgrammingWorld package*. This package represents the meta-model of a concrete MPW. It is oriented to the design of the PSE-Simulator (Section 3.1.1) and the Solist meta-model (Section 3.1.2). Figure 4.6 shows the modeled Ecore package of the MPW package.

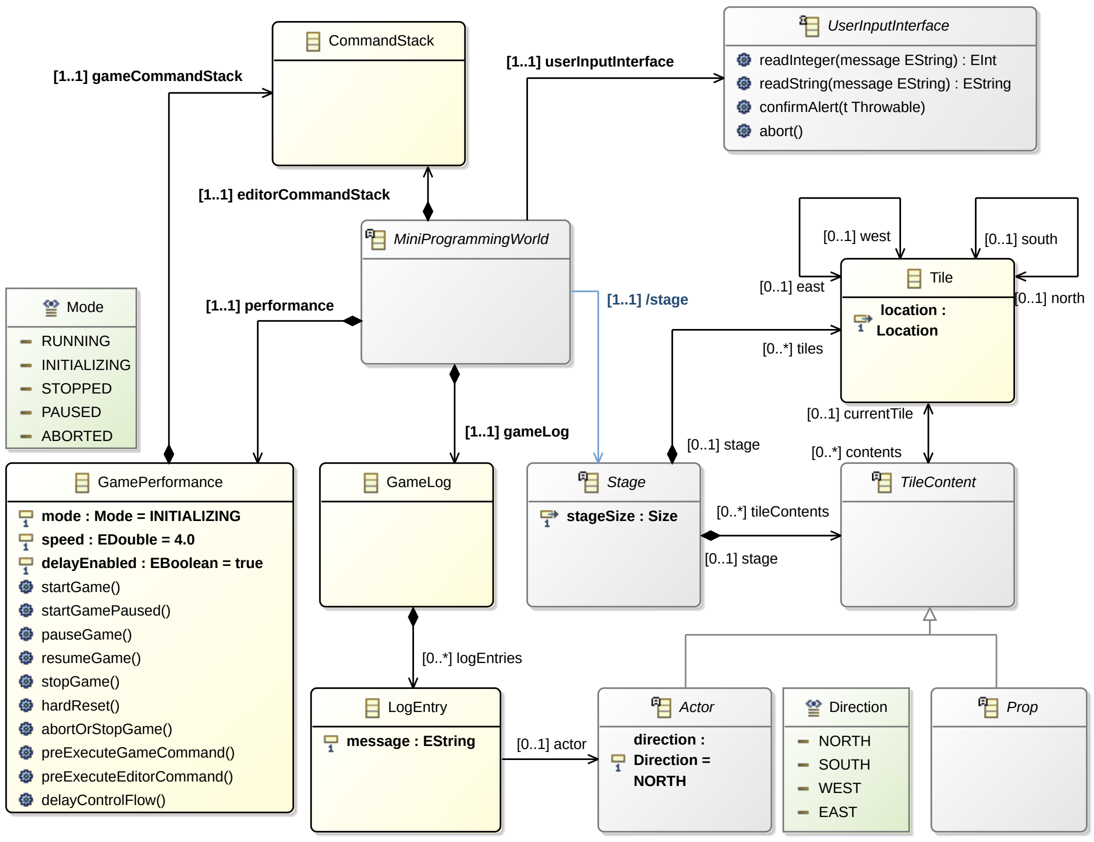

Figure 4.6: MiniProgrammingWorld package

`MiniProgrammingWorld` acts as a root container and represents an abstract meta-type, which is implemented by concrete game classes like `HamsterGame` or `KaraGame`. Important composed classes are `GamePerformance`, `Stage` and `GameLog`. There are two instances of `CommandStack` imported from the *Command package*. The `editorCommandStack` is used for editor commands, while `gameCommandStack` executes game commands and is controlled by the `GamePerformance`. The link between `MiniProgrammingWorld` and `Stage` is marked as `derived` indicated as a blue reference. This implies that the concrete containment reference is set in the derived model. For example, in the hamster simulator the `HamsterGame` defines a `territory` reference to the `Territory`, which is used as a replacement for this derived `stage` reference on the meta-level.

The class `Stage` represents the meta-type for concrete two-dimensional maps like the hamster simulator's territory. Its dimensions are defined by the attribute `stageSize`, which makes use of the helper type `Size`. Figure 4.7 illustrates an example of the contents of a stage, which consists of two rows and two columns. A stage consists of multiple tile-instances of type `Tile`, which are contained in the `tiles` reference. Each tile has a location consisting of its column and row indices. Important references are the four bidirectional links `east`, `west`, `south` and `north`, which are used to build up a mesh. Whenever tiles are neighbors of each other, these links are set accordingly. The connections by these links are used to let actors move from tile to tile in this mesh. Besides tiles, a stage acts also as a container for objects of type `TileContent`, which can be placed on tiles as contents. A `TileContent` is semantically placed on a tile, if the bidirectional reference `currentTile` is set and hence it is contained in the `contents` references of the `Tile`.

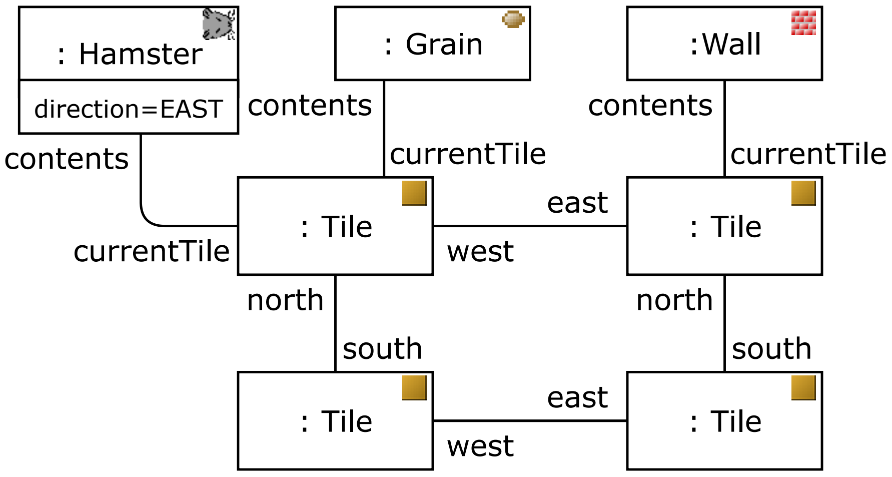

Figure 4.7: Example object diagram of tiles and tile contents

There are two sub types of `TileContent`: `Actor` and `Prop`. `Actor` defines the meta-type for objects which are controlled by clients, like the hamster in the hamster simulator or the ladybug in *Kara the ladybug*. A special property is its `direction` allowing it to look to one of the four directions `EAST`, `WEST`, `SOUTH` or `NORTH`. `Prop` is the second sub type of `TileContent`, which is used for any requisite to be placed on the stage. They are not intended to have a direction nor to accept any command to let it actively move to another tile. Examples are grains or walls in the hamster simulator. In *Kara the ladybug*, other examples are leafs, trees and mushrooms.

Similar to the class `Performance` of the Solist design, the class `GamePerformance` controls the state of the game and handles suspending of the control flow. An important state variable is the field `mode` of enumeration type `Mode`, which can take one of five values. Several methods provided by `GamePerformance` are used to realize a state machine, which is depicted in Figure 4.8. The label "<any>" is used to indicate, that the transition can be executed from any of the five states. Transitions, which are not explicitly depicted, have no changing effect in the `mode` variable. On creation of a new game, the state machine starts in state `INITIALIZING`. From this state usually the game is started, either with `startGame()` or `startGamePaused()`. While the former switches to the state `RUNNING`, the latter lets the game start in `PAUSED`. The state `RUNNING` indicates that the current game is processing a client program and processes game commands. As a restriction, undo and redo actions are not allowed to be called in the running state. From the running state, it is also possible to switch into `PAUSED` by calling `pauseGame()`. The paused state is used to suspend the client program, which e.g. has the effect, that the hamster does not move. Additionally, in the `PAUSED` state undo and redo actions are allowed. By calling `resumeGame()`, the state `PAUSED` can be leaved. There are two states which define that a game is stopped. The first one is `STOPPED`, which gets active whenever `stopGame()` is called or an exception is thrown while executing a game command. `STOPPED` allows undo and redo of commands, but not to process new game commands. The second stopped mode is `ABORTED`, which occurs when calling `abortOrStopGame()` in one of the two states `RUNNING` or `PAUSED`. By executing a game command in state `ABORTED`, a transition to `STOPPED` is processed and also a dedicated `GameAbortedException` will be thrown. From any state, the method `hardReset()` can be used to set the state back to `INITIALIZING`, where a new stage can be created. Editor commands are only allowed to be processed in `INITIALIZING`.

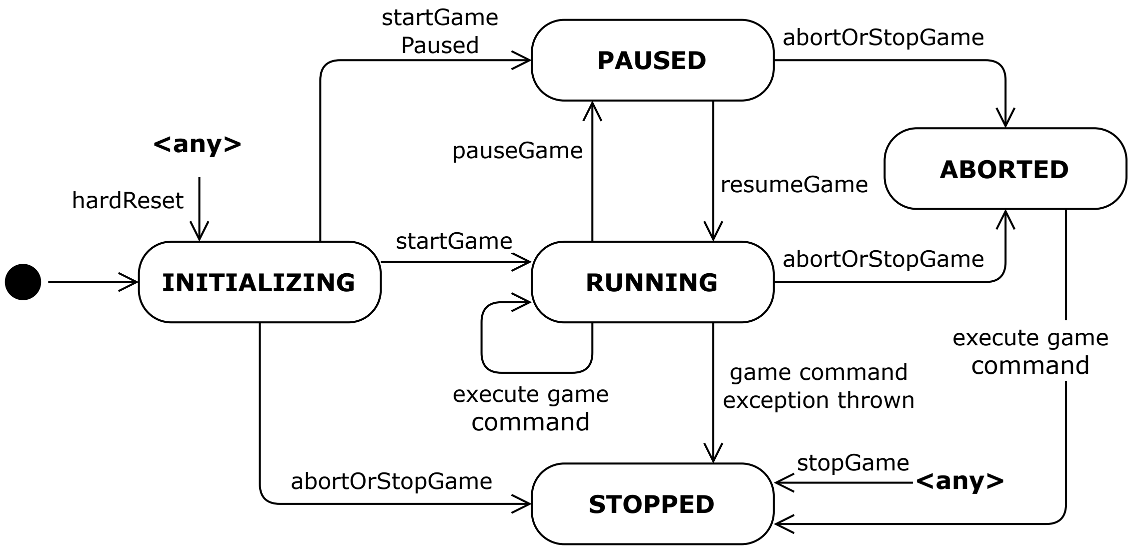

Figure 4.8: GamePerformance state machine

Another class of a MPW is `GameLog`, which is used to collect a list of `LogEntry` objects. Each game command is generated to insert a message into the game log when executed. Additionally, it notices the related actor of the game command by setting the `actor` reference of the related log entry. This information can be used to mark log messages e.g. by color, to distinguish them between multiple actors. The text message itself is set with the attribute `message`.

To be able to call operations back to a higher layer in the architecture, a `UserInputInterface` is hold as a dependency of `MiniProgrammingWorld`. It is injected using the *Dependency Injection pattern* [Fow04] and supports callbacks to read integers, read strings or confirming alerts. This interface is usually implemented in a UI framework component to get feedback from the user. When invoking `readInteger()` or `readString()`, the user is asked to type strings or integers into a text-box. Exceptions, which occur during the simulation, like violating preconditions, are shown as an alert dialog box when calling `confirmAlert()`. For testing, the `UserInputInterface` can be replaced by a test double like a mock or fake implementation.

4.5.4 Roles Idea

Like the design of the PSE-Simulator, for stages and actors like `Territory` or `Hamster` separate classes are designed for the representation of roles. Basically, there are two kinds of commands: editor commands and game commands. Additionally, information like the current tile of a tile content might be gathered. This leads to three roles of an actor or a stage: *game*, *editor* and *read-only*.

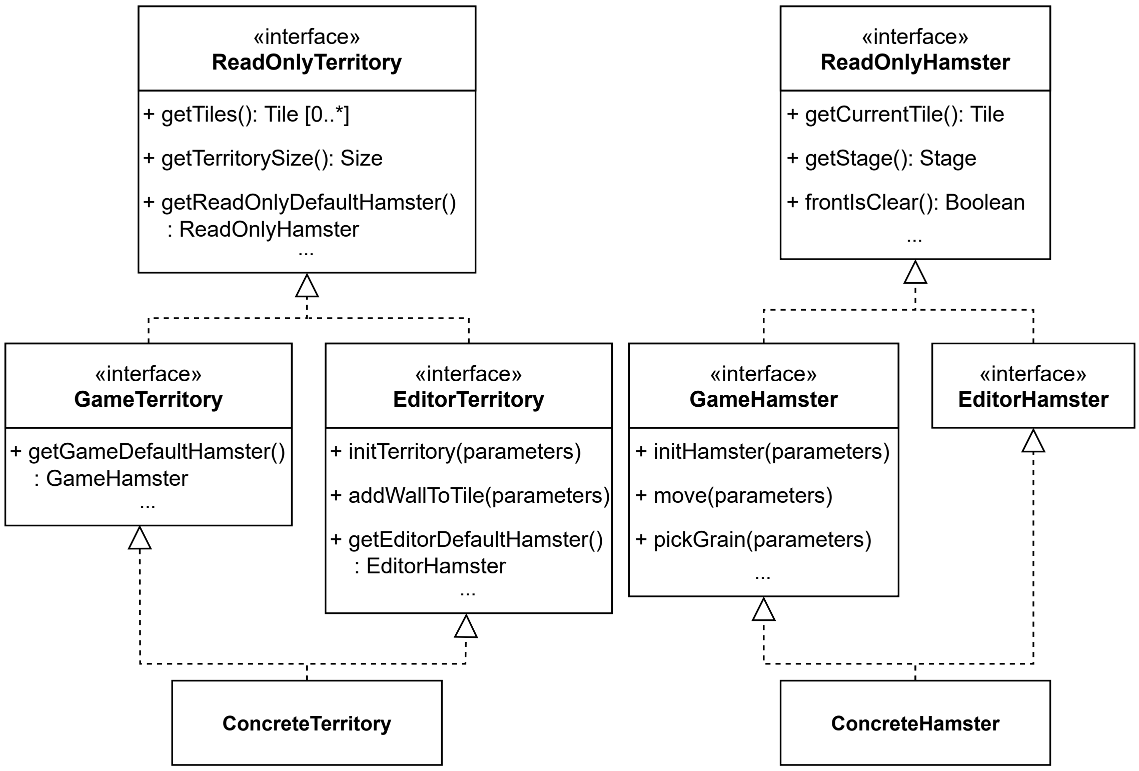
«interface»
ReadOnlyTerritory
+ getTiles(): Tile [0..*]
+ getTerritorySize(): Size
+ getReadOnlyDefaultHamster() : ReadOnlyHamster
...
«interface»
ReadOnlyHamster
+ getCurrentTile(): Tile
+ getStage(): Stage
+ frontIsClear(): Boolean
...
«interface»
GameTerritory
+ getGameDefaultHamster() : GameHamster
...
«interface»
EditorTerritory
+ initTerritory(parameters)
+ addWallToTile(parameters)
+ getEditorDefaultHamster() : EditorHamster
...
«interface»
GameHamster
+ initHamster(parameters)
+ move(parameters)
+ pickGrain(parameters)
...
«interface»
EditorHamster
ConcreteTerritory
ConcreteHamster

Figure 4.9: Roles design

In contrast to the PSE-Simulator design, this is not solved by creating a hierarchy of concrete classes. Instead in this project, interfaces are used to get a diamond inheritance, which is depicted in Figure 4.9. On the left side, the example of the `Territory`'s role is shown. The base interface is `ReadOnlyTerritory`, which represents the read-only role. It provides methods for gathering any property of the territory, like obtaining the tiles collection, the territory size or the read-only role of the default-hamster. Additionally, further queries like `isLocationInTerritory()` can be modeled for the territory, which are also declared by this interface. `GameTerritory` and `EditorTerritory` are extending the read-only role interface. While there are usually no game commands modeled on a stage, this game-role does not define further command operations. But as an extension to its read-only role, it provides a getter-operation which returns the game role of the default-hamster. Editor commands modeled for the territory are declared on the `EditorTerritory` interface. Examples are `initTerritory()` or `addWallToTile()`. Each command provides an operation and takes additional arguments which are placed inside a single `parameters` object. Similar to the other two roles, the `EditorTerritory` interface defines a getter-operation which returns the editor role of the default-hamster. Finally, there is one concrete implementation class for all three roles, which is named with the prefix "Concrete". These concrete implementation classes are completely generated. In case of territory, the concrete class is `ConcreteTerritory`, which implements all commands, queries and getter-operations defined by the three roles.

In addition to the territory roles as a representative example of a stage, the example of the hamster is shown on the right side of Figure 4.9. Symmetrically to the territory, there are four classes which make up a diamond inheritance. `ReadOnlyHamster` defines the getter-operations for properties and queries modeled for the hamster. `EditorHamster` is empty, since there are no modeled editor commands, therefore it defines no further details to its read-only role. `GameHamster` instead defines all operations of the hamster commands, like `move()` or `pickGrain()`. To allow that multiple hamsters are placed on a territory, there is an `initHamster()` game command. This command allows to be executed while the game is running and initializes a further hamster. Finally, the class `ConcreteHamster` implements all three role interfaces.

4.5.5 View Model Design

This section describes the package, which contains the Ecore types used to define the view model. Figure 4.10 shows the class diagram of the Ecore package. While the `GameViewPresenter` represents the UI logic, the class `GameViewModel` and its related types represent an observable data-structure.

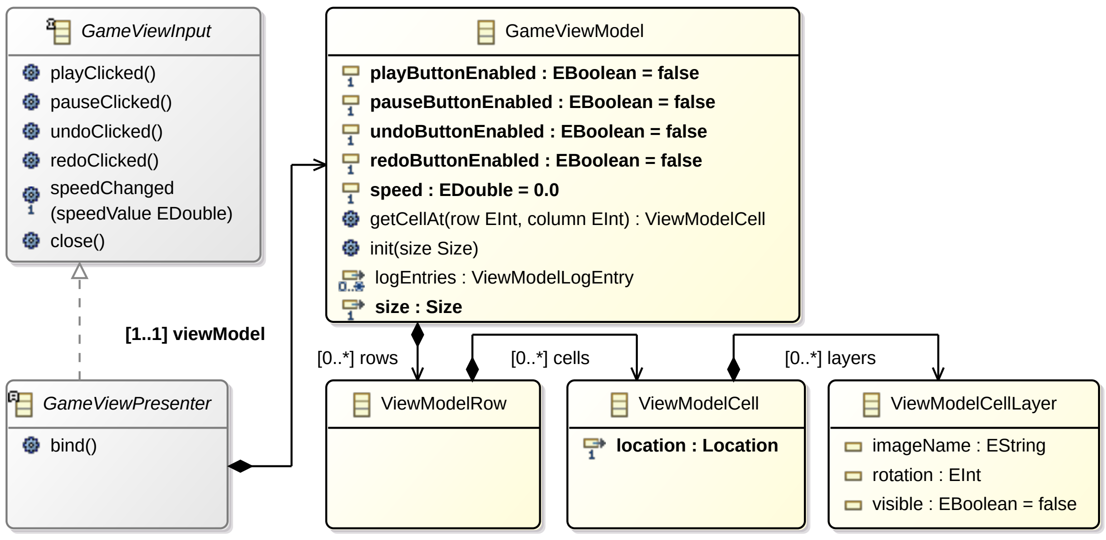

Figure 4.10: View model package

`GameViewPresenter` implements the interface `GameViewInput`, which defines several input operations used to perform interactions with the UI. The first four operations `playClicked()`, `pauseClicked()`, `undoClicked()` and `redoClicked()` are event-handlers for clicking on buttons. Further, the input operation `speedChanged()` is used to change the delay, which is inserted between game commands. `GameViewPresenter` implements these operations by delegating to the `GamePerformance` class. Finally, there is a `close()` input, which indicates that the user wants to close the application. Besides derived input operations from `GameViewInput`, the presenter defines a `bind()` operation. This operation contains most of the UI logic, since it picks the relevant parts of the core model and adds observers to them. Using these observers, the view model will be updated accordingly. For common UI logic, the MPW simulator framework includes a `GameViewPresenterBase` class derived from `GameViewPresenter`, which can be derived by concrete presenter classes.

The view model itself is defined by the class `GameViewModel`, which provides any information used to control the visible state of the UI. It defines four boolean properties `playButtonEnabled`, `pauseButtonEnabled`, `undoButtonEnabled` and `redoButtonEnabled`. Each of those define if the related button shall be enabled. The `speed` property can be used to display the current speed value, e.g. by a slider-control. The helper operation `getCellAt()` is used to return the related `ViewModelCell` object for a given row and column position. With `init()`, the `GameViewModel` takes a `Size` and creates the given amount of `ViewModelRow` and `ViewModelCell` objects. Further, it stores the dimensions in the property `size`.

The main content of the view model is defined by the `ViewModelRow` references. Each `ViewModelRow` is representing a row of the stage. While a row of a stage consists of `Tile` objects, in the view model `ViewModelCell` objects are used, which are stored in the `cells` containment reference. Therefore, each `ViewModelRow` has to contain one `ViewModelCell` for each column of the stage. The contents of each cell in the view model are defined by objects of type `ViewModelCellLayer`. Typically `TileContent` objects are mapped to these layers, which are ordered and represent images to be rendered. Since the view model has a clear focus on the information to be rendered on screen, it contains a logical name of the target image in the attribute `imageName`. Further, it defines the rotation in degrees and a boolean flag, indicating if the layer shall be hidden or visible. Since these information details are filled by the presenter, the UI framework specific part has no special logic regarding rotation, ordering or selecting the images for a cell. Besides rows, cells and layers, the view model also defines the list of log entries. They are stored in the containment reference `logEntries` and are represented by objects of type `ViewModelLogEntry`, which defines a `message` and a `color`. These entries are intended to be displayed in a list of strings, which can be colorized to indicate further information like which hamster has done the relating action.

4.6 Testing Strategy

This section describes the approach for testing code of the proposed solution. Testing is the fundamental method used in this project to ensure the correctness of the software and enable to safely refactor code. Like described in Section 4.2, there are multiple steps involved to develop the MPW modeling workflow. Each of the three major workflow steps is tested in the following way:

- Modeling: The first step in the workflow is the modeling of entity models and operational behavior. On the one side, the validation of the modeling tools are used to ensure, that the modeled elements are syntactically correct. On the other side, an advanced validation is done by processing OCL rules after loading these models. For example, these OCL rules ensure that the modeled Henshin commands fulfill certain assumptions and do only use features, which are supported by the model-to-model transformation. While these validations are only covering static aspects of the modeled artifacts, they represent a kind of testing to early catch potential errors.

- Transformation: The transformation of the modeled artifacts covers dynamic aspects, which is tested differently. For this project, there are two ways regarded, which can be used to test the transformation aspects: directly or indirectly. In a direct test approach, transformation logic to transform given inputs can be tested in an isolated way, such as by checking the immediate output models. In contrast, an indirect approach does not test transformations directly, but e.g. focuses on the generated code, which results by these transformations. In this project, the direct approach is explicitly avoided. Since one disadvantage is a more difficult later refactoring. This is caused by the fact, that the transformation structures are bound by tests. An indirect approach is much better suited here. First, the compiler represents a first process, which can be used to easily ensure that the basic transformation logic is valid. Second, testing against the generated code can be used to cover transformation logic indirectly, while not forcing to couple test code to transformation details.
- Generated Simulator Code: The last step in the workflow produces simulator code, which is used to get an executable MPW simulator. The generated simulator code is intensively tested using unit tests on two abstraction layers. On the lower abstraction layer, tests are directly written against the simulator's core model, e.g. by using the facade or internal classes. By using the *Humble Object pattern*, the view model additionally provides a second abstraction layer to test the simulator code. On this layer, also several tests are written to ensure that the presenter code works integrated with the simulator's core. By using this approach, a high code coverage can be achieved, which automatically ensures that the transformation and modeling steps have to be correct.

Since this project is developed in an iterative approach, the transformation and generation code has been refactored multiple times. These refactoring actions have been performed safely, since the high coverage of the simulator code ensured, that the simulator's API and functionality were not broken. By using more unit tests to directly test transformation and generation code, these refactoring actions probably would have required much more effort.

4.7 Documentation

In this section, different approaches to document the MPW simulators are depicted. First, the usage of MDSD in this project already provides diagrams for several meta-models, which are useful to document the architecture and design aspects. In Ecore models, the annotation ”`http://www.eclipse.org/emf/2002/GenModel`” is used to add a `documentation` entry on `EModelElement` instances. On the one side, this documentation is displayed in the Ecore editor of Eclipse, while on the other side it is used to be generated as documentation comments in the target programming languages. Further, Design by Contract focuses on a formalized way to document constraints on operations, which is provided in this project by the usage of the Query-DSL (Section 5.1.3). Invariants, pre- and postconditions are used to generate documentation comments on the generated operations. Additionally, they can be documented by comments in the Query-DSL themselves. These comments are generated as simple comments in the related statements in the executable code. Manually written code, e.g. in QVT-O, Xpand, Xtend, Java or C++ is documented by appropriate comments which are supported by the related programming language.

Besides the explicit documentation, several unit tests are written in a behavior-driven manner. They follow a pattern of the wording *given*, *when* and *then*, which specify the arrangement, acting and assertion of the test. With this approach, unit tests can also be seen as a formalized documentation of the working simulators.

Finally, an overall documentation of the whole project is done in a wiki of the related code repository. It describes architectural concepts and design decisions related to the development of this thesis.

4.8 Adaptability

The non-functional requirement to adapt the hamster simulator for other programming languages is one of the main objectives in this work. Therefore, the first subsection 4.8.1 gives an idea of how an adaptation of the hamster simulator can be done for other programming languages. As a starting point for the adaption, the Java based hamster simulator is used, which has been re-implemented in this thesis. Another goal of this thesis is the adaptability to other MPWs like Kara the ladybug, which is handled by the second subsection 4.8.2.

4.8.1 Adding new Programming Languages

This subsection states a workflow, how the hamster simulator can be adapted for further programming languages. Like mentioned before, the Java based simulator is used as a starting point, which already brings a working code generator, multiple test-cases and a helpful orientation of a possible implementation.

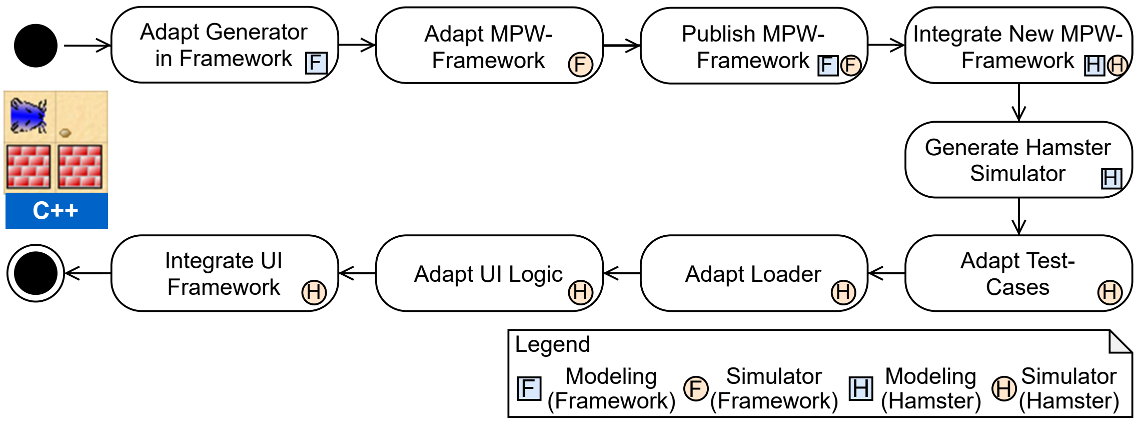

Figure 4.11: Adaption for new programming languages

Figure 4.11 illustrates the workflow used in this thesis to adapt the hamster simulator for further programming languages, like C++. Steps are performed in the *modeling* or *simulator* environment (see Section 4.2), indicated by a small icon. The first step is the adaption of the code generator, written in Xpand. Section 5.3 describes in more detail, how the code generator templates are structured. However, the adaption of these code generator templates is performed in the following steps:

1. Entity Templates: First, the Xpand templates are adapted, which are responsible for generating the entity model. Different types of `EClassifiers` like `EClass` and `EEnum` are handled to be generated correctly for the target programming language. Furthermore, the `interface` flag on `EClass` has to be regarded to generate interfaces aligned with the target language. Additionally, there are custom stereo-types like `ValueType` defined, which is used to generate an `EClass` as a simple value type like `Location` or `Size`. An important aspect that also needs to be considered is the life-cycle-management of objects. For C++ as an unmanaged language, entity compositions are realized using smart-pointers. Simple references, which have no containment semantics, are instead mapped to weak-pointers. Further, a generic access to properties is required, which is solved in Java by reflection. In C++ instead, an own reflection mechanism is required to be generated for entities. Finally, observable properties in generated entities have to be regarded.
2. Query Templates: The second templates to adapt are used to generate expressions for queries and constraints. The adaption is relatively straight forward, since expressions or statements, like defining a variable, are easily mapped to other programming languages.
3. Command Templates: Afterwards, the templates for graph transformations in commands are adapted. Like the templates for queries, parts of generating commands can be transferred to other languages by a mapping of each statements to the target programming language feature.

As a short, intermediary summary of the experience gathered by adapting the code generator from Java to C++, the most effort is spent into the entity generation. This is caused by a different support of programming language features on class and object level, like an unmanaged life-cycle management, the need for observable properties or missing reflection capabilities. These missing features are implemented manually or solved by adapting the generator appropriately. There can be assumed, that adapting to other programming languages is more straight-forward, since Java and C++ templates are provided as a result of this project. They already show, how these features can be generated by example.

The second main step of the workflow in Figure 4.11 is about manually adapting the MPW framework. By executing the adapted generator from step one, several stubs are already pre-generated. Especially primitive commands `AddEntityCommandImpl`, `RemoveEntityCommandImpl` and `SetPropertyCommandImpl` have to be implemented. Also, central classes like `GamePerformanceImpl` or `CommandStackImpl` needs to be adapted accordingly. In addition, the base class `GameViewPresenterBase` for concrete presenters and special methods in the `GameViewModelImpl` have to be adapted. Optionally, helper classes like central assertions or exception types can be defined as part of the framework.

After adapting the MPW framework, it has to be published in an appropriate way to allow concrete MPWs to depend on it. For Java, this publish step is processed by Maven and a public Maven Repository. In C++, the publishing step is skipped, since a method is used, where library code gets re-compiled by the library user. CMake is used to download the C++ code and compile it with the same compiler settings like the target MPW simulator environment. For other languages, an appropriate method has to be applied to deploy a library for later reuse by other repositories.

The next steps take place in the hamster simulator environment as a concrete MPW, where several commands and queries are already modeled. First, in the modeling environment of the hamster simulator, the new version of the MPW modeling framework needs to be integrated. This will extend the hamster modeling environment by the new programming language generator. In the hamster simulator core, the MPW simulator framework library has to be integrated, too. Next, the generation has to be executed, which will generate source code for the simulator in the target programming language. Based on the generated core, it is suggested to adapt unit tests of the existing hamster simulator to get a reliable test harness. If these tests pass, the `TerritoryLoader` shall be implemented as a next step. It is responsible to parse territories from input streams and files. Accessing streams has to be adapted to related features based on the target programming language.

The last two steps are about the UI. First, the logic has to be adapted, where primarily the concrete presenter of the hamster simulator has to be implemented. Afterwards, an appropriate UI framework has to be chosen, similar to JavaFX for Java or SDL for C++. Intended by the separation of the view model data-structure, this code is relatively simple. In case of a rendering loop, which is the case by game-frameworks like the SDL, in each loop iteration the view model contents are mapped to rendering calls. Otherwise, some frameworks like JavaFX are based on UI components, where the view model is observed for changes and related UI components are updated accordingly.

It is important to notice, that this workflow does not imply a strict sequential flow. At any step, it may require to go back to a previous step e.g. in case of a compile error caused by mistakes in the adapted generator. Therefore, it might be helpful to split some steps into separate concerns like the entity model, commands or queries and perform iterations based on them. To allow a more efficient loop between the hamster and framework steps, the publishing might be performed on the local machine. Finally, other derivations of this workflow might be reasonable. In summary, the adaption to a new programming language comes with some effort, especially when language aspects like life-cycle-management are different to previous ones. But this effort primarily has to spent only once, while allowing that multiple MPWs can be generated on the new target programming language.

4.8.2 Adding new Mini-Programming-Worlds

As the second requirement on adaptability, the MPW framework shall provide a base to model further MPWs as well. This subsection therefore describes, how to adapt a new MPW like Kara the ladybug. Figure 4.12 gives an overview over the workflow, which shows the major steps for this adaption. While the first four steps are performed in the concrete MPW modeling environment, the other ones are primarily done directly in the concrete MPW simulator environment (see Section 4.2).

The first step is to model the concrete entities for the new MPW (see Section 5.1.1). Similar to the hamster simulator entity model, an Ecore model is created which is based on the MPW meta-model. In case of Kara, the following entity types are modeled:

- `Kara`: Derives from the meta-type `Actor`, similar to the `Hamster`.
- `World`: Extends from `Stage` and provides the two-dimensional world. It is similar to the `Territory` of the hamster simulator.
- `Tree`, `Mushroom` and `Leaf`: Three different types deriving from `Prop`. While trees and mushrooms block the ladybug from moving, leafs can be collected or put on tiles.

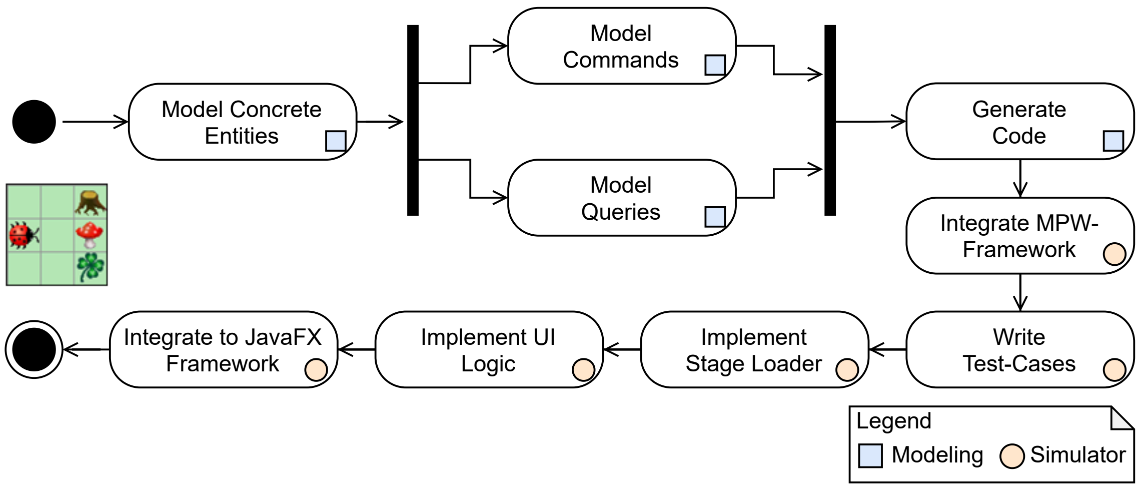

Figure 4.12: Adaption for new MPWs

The next two steps are about modeling the commands and queries of the MPW. Game or editor commands are modeled in Henshin (see Section 5.1.2), while it might be helpful to use the existing hamster game commands as an orientation. For example, the relatively complex `initTerritory()` editor command of the hamster simulator can be very closely mapped into an `initWorld()` editor command for the Kara MPW. As an example of the Kara MPW, the following game and editor commands are modeled:

- `move()`: Moves Kara one tile in front of its facing direction. Additionally, a mushroom in front of the actor is also moved, if the tile behind the mushroom is clear.
- `removeLeaf()` and `putLeaf()`: Removes a leaf on the tile of Kara or puts a leaf on the tile.
- `turnLeft()` and `turnRight()`: Turns the actor to the left or right direction.
- `putTreeToTile()`, `putMushroomToTile()` and `putLeafToTile()`: Editor commands to put props on a given tile.
- `clearTile()`: An editor command, which clears any contents from a given tile.
- `initKara()` and `initWorld()`: Two editor commands which are used to initialize the actor instance `Kara` or stage instance `World`.

In parallel to modeling commands, the queries and constraints of the MPW can be modeled. Like modeling the hamster simulator, these elements are modeled using the Query-DSL (see Section 5.1.3). Several queries are also similar to the hamster queries. Again, in case of the Kara MPW, some example queries are the following:

- `mushroomFront()`: Checks if the tile in front of Kara contains a mushroom.
- `onLeaf()`: Checks if the current tile of Kara contains a leaf.
- `treeFront()`, `treeLeft()` and `treeRight()`: Determines if on the left, right or in front of Kara a tree is placed.

Besides queries to gather information, also several constraints for preconditions, postconditions and invariants are modeled. Finally, after all required commands and queries are modeled, the code generator is executed.

The next steps in the workflow are handled in the simulator's programming environment for the language Java. The adaption for other programming languages like C++ is performed similar. First, the programming environment has to be set up and the MPW framework library has to be integrated. In case of Java, primarily Maven build scripts are used, which are close to the hamster simulator ones.

After the setup, it is intended to write several test-cases for the modeled commands and queries. On the one side, this can be done directly on the MPW facade. On the other side, they can be written on the level of the view model. It is also intended to adapt the test utilities and similar test-cases from the hamster simulator.

Based on these tests, the missing stage loader class is implemented. For Kara, the class is named `WorldLoader` and internally uses the generated `WorldBuilder` class. The loader class can also be adapted from the hamster simulator's `TerritoryLoader`, as most of the implementation is very close. Since the loader deals with a serialized representation of the stage, every prop can be mapped to a character. If the serialization is much more complex compared to the hamster simulator, it has to be implemented by custom logic, which is not scope of this thesis.

After implementing the stage loader class, the presenter has to be implemented. In case of Kara, the presenter class will be named `KaraGameViewPresenter`. In the MPW framework, an appropriate base class is intended to be reused, which already handles common logic. It remains to implement the logic to fill the view model cells with related `ViewModelCellLayer` objects. Typically this is done by observing the contents of the stage and inserting the layers into the cells, which have the same location.

As a final step, the simulator's UI and core parts will be integrated into the UI framework. In case of Java, most of JavaFX framework related code of the hamster simulator can be re-used. Additionally, the images of the custom MPW have to be included by their related names.

Like the adaption to a new programming language described in Section 4.8.1, the presented workflow is not strictly required to be done in this sequential order. For example, the test-cases can be written after implementing the UI logic, but describing them first enables a test-first approach. Performing the workflow in iterations is also reasonable.

4.9 Tooling and Technologies

This section describes, which tools are used for modeling and implementing the MPW simulators in this project. Like already highlighted in previous chapters, Eclipse EMF[8] is used as the base platform.

[8] `https://www.eclipse.org/`

First, the used tools for modeling and transformation are described. Entity models are defined using Ecore and a visual representation is also provided by Ecore diagrams, which are stored by `.aird` files. For deploying Eclipse plugins, the OSGi[9] standard is used, which enables dependency management based on a dedicated repository layout. Henshin[10] is used to visually model game and editor commands as graph transformations for concrete simulators. Queries are modeled by the custom Query-DSL, which is developed using the Xtext framework[11]. Validation of models is primarily performed by using the OCL Software Development Kit (SDK)[12]. Model-To-Model transformations are realized using QVT-O. QVT-O allows to use Ecore models, query models and Henshin models as input and operates on these models. For code generation, Xpand[13] is applied, which allows to use text templates to generate code. As a workflow language to integrate these tools, the MWE2[14] is used. MWE2 allows to call custom components written in Java. This aspect is used to call e.g. the Java API of QVT-O to integrate other technologies into a MWE2 workflow. While formatting generated code in Java can be done by a pretty-printer provided by Xtext, for C++ the Eclipse C/C++ Development Tooling (CDT)[15] is used, which provides a separate pretty-printer to format C++ code.

The source code of this project is versioned with Git and stored on GitHub[16], which also provides a *continuous integration* workflow. To execute automated tests and use the fast feedback of continuous integration, the modeling workflow is also managed by the build tool Maven[17]. Since most of the modules are based on Eclipse, the Maven plugin Tycho[18] is applied as an extension.

Finally, concrete MPW simulators are built and executed independently of modeling tools. For Java, the build tool Maven is also used, but without using Tycho. The UI of Java simulators is based on JavaFX[19]. Automated testing is performed with JUnit 5[20]. For C++, MPW simulators are managed by the cross-platform build tool CMake[21]. CMake allows to build C++ on multiple tool chains based on different operating systems like Windows, MacOS or Linux. Rendering is performed by the SDL 2[22], which provides a platform independent rendering API. GoogleTest[23] is used to execute automated tests for C++.

[9]https://www.osgi.org/
[10]https://www.eclipse.org/henshin/
[11]https://www.eclipse.org/Xtext/
[12]https://projects.eclipse.org/projects/modeling.mdt.ocl
[13]https://projects.eclipse.org/projects/modeling.m2t.xpand
[14]https://www.eclipse.org/Xtext/documentation/306_mwe2.html
[15]https://www.eclipse.org/cdt/
[16]https://github.com/
[17]https://maven.apache.org/
[18]https://www.eclipse.org/tycho/
[19]https://openjfx.io/
[20]https://junit.org/junit5/
[21]https://cmake.org/
[22]https://www.libsdl.org/
[23]https://github.com/google/googletest

5 Modeling Workflow

This chapter is about concepts used to implement the modeling environment, which represents the MDSD approach used to develop the project. While Chapter 4 has the focus about general concepts, the design of meta-models and the architecture of MPW simulators, this chapter describes the workflow with its transformations on the modeling side. Figure 5.1 gives an overview of the three phases *modeling*, *transformation* and *generation*, which are outlining the workflow. Additionally, the relevant artifacts are shown, which are related to each phase.

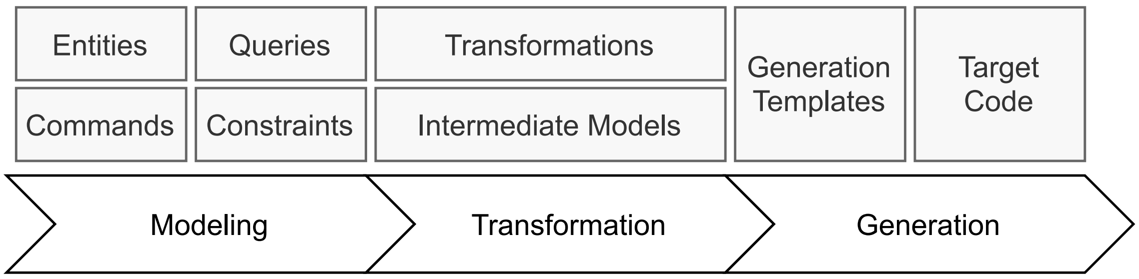

Figure 5.1: Overview of the MDSD workflow

First, in Section 5.1 the input modeling will be illustrated, which cover the modeling of Ecore entities, Henshin commands, Query-DSL queries and constraints. Second, in Section 5.2 the essential logic of the model-driven workflow is shown, which is implemented by model-to-model transformations. The transformations cover further validation of the inputs, intermediate models adjusted for the final code generation and also the operational logic. These transformations are written in QVT-O to transform these models. Last, in Section 5.3 an overview is given for the generation of target code like Java and C++, based on Xpand generation templates. The generation is described separately for the generation of entities, queries and commands.

5.1 Input Modeling

This section deals with the modeling of artifacts, which are used as input for the transformation phase. The following subsections will describe, how these models are created using Ecore, Henshin and the Query-DSL. In the following, the hamster simulator is used as a concrete example of a MPW, while the ideas can be mapped to other MPWs as well.

Figure 5.2 illustrates the relevant artifacts, which consist of entities, commands, queries and constraints. The first activities are to define the MPW meta-models and the concrete hamster meta-model. Both activities provide Ecore models as a result. Next, these meta-models are used to write queries and model Henshin commands. Additionally, constraints for the modeled commands are written, which specify pre- and postconditions. As a result, Henshin and Query-DSL based artifacts are produced, which in addition to the Ecore models represent the outcome of the modeling phase.

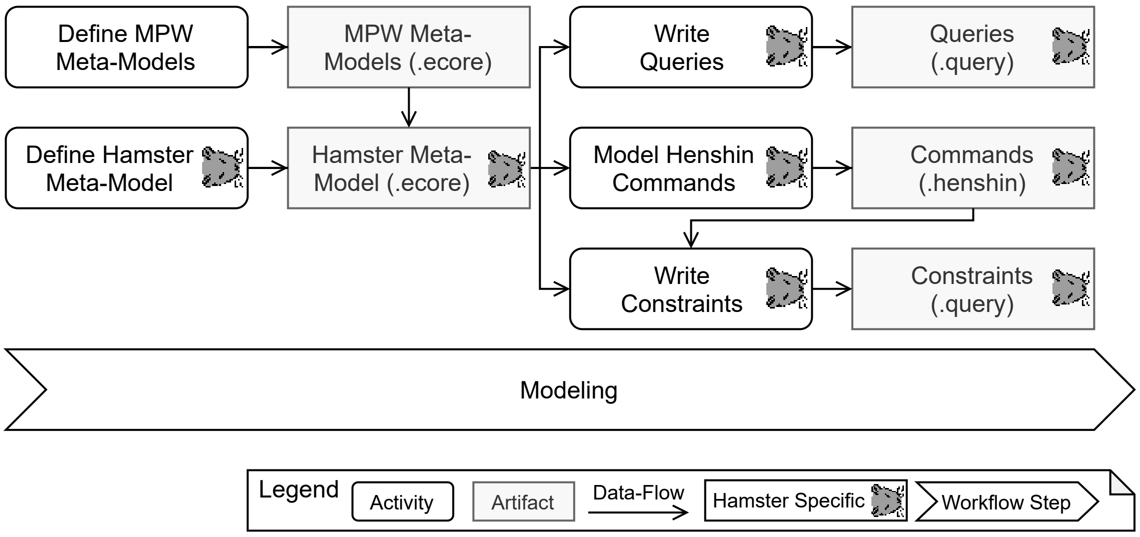

Figure 5.2: Modeling phase of the MDSD workflow

5.1.1 Ecore Entities

As input artifacts for the workflow, the MPW meta-models are introduced already in Chapter 4 for illustrating the concept of an abstract design to describe MPWs. This subsection further illustrates how the hamster model based on the *MiniProgrammingWorld* package is designed. Figure 5.3 shows the Ecore diagram of the *Hamster package*. The class `HamsterGame` derives from the meta-class `MiniProgrammingWorld` to specify the concrete root class of the game. `Territory` acts as the concrete realization of the `Stage`, which is contained in the game class by a containment reference `territory`. The figure also shows the derived `stage` reference between the meta-classes `MiniProgrammingWorld` and `Stage`, which is realized by using the concrete `territory` reference. `Hamster` is defined as the `Actor` class of the hamster simulator. Since a hamster can pick grains into its mouth, it defines a collection reference named `grains`, which can hold `Grain` objects. The territory defines a dedicated default hamster, which is realized by the reference `defaultHamster` and is intended to be used in simple scenarios, e.g. where only one hamster is needed. Additionally, further `Hamster` objects can be created, which will be added to the `tileContents` collection of the `Stage` parent type. `Grain` and `Wall` inherit from `Prop` and define further contents, which can be placed on tiles of the territory. While grains are used to be picked and placed by game commands called on a `Hamster` object, walls do block the relating tile and are only intended to be placed or removed by editor commands.

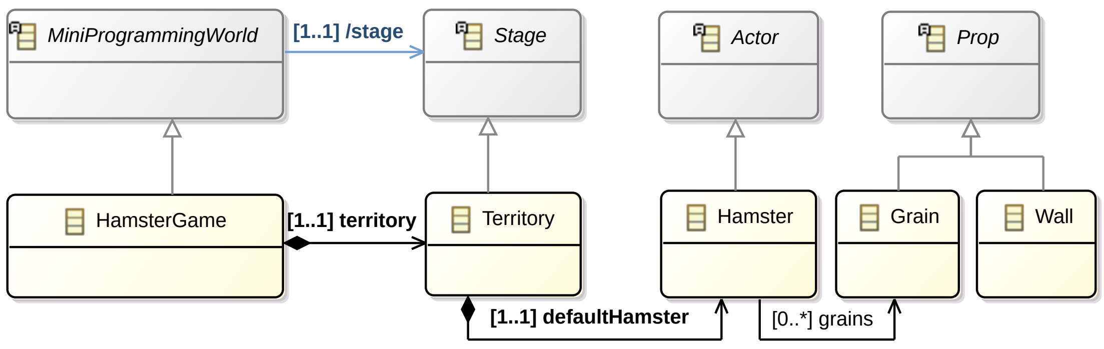

Figure 5.3: Hamster package

5.1.2 Henshin Commands

This subsection outlines the modeling of Henshin modules, which express game and editor commands and are used as input artifacts for the transformation phase. Henshin allows to model graph replacement rules, which declaratively describe, how an object graph will be modified for a certain rule. The editor of Henshin can be reused to visually draw these graph replacement rules. Figure 5.4 depicts the `move()` command of the hamster simulator as an example.

Since most of commands needed for MPWs are relatively simple, not all flexibility provided by Henshin is needed. The following restrictions are defined for modeling MPWs with Henshin:

- Each graph replacement rule has to define one object with the name `self`, which is typed for the target `Actor` or `Stage`. This is used to have an explicit starting point, which makes the generation of related code easier.
- Starting from the object named `self`, every other object of the left side graph has to be reachable by traversing existing edges. This is a consequence of the previous restriction, since only `self` is used as a starting point. Alternatively, given input parameters are also considered as possible starting points.
- When defining any control unit like a sequence, which is no rule, one of these control units has to be defined as the main unit. To define a main unit, it has to be named equal to the Henshin module's name.
- If there is no main unit explicitly defined, each rule has to be distinguishable by attribute conditions on the `self` object. For example, the move command depicted in Figure 5.4 defines a `direction` condition for each of the four rules. The resulting command is like a `PriorityUnit` in Henshin, where each rule represents a possible case to be matched.

These restrictions are validated by OCL expressions, which are checked at the beginning of the transformation phase. By using Henshin control units of types like *ConditionalUnit*, *IteratedUnit*, *PriorityUnit* or *SequentialUnit*, also commands with higher complexity can be modeled. This is e.g. used for the `initTerritory()` command, which has to loop over two dimensions to create the related `Tile` objects. Additionally, these objects have to be connected consistently with their neighbors.

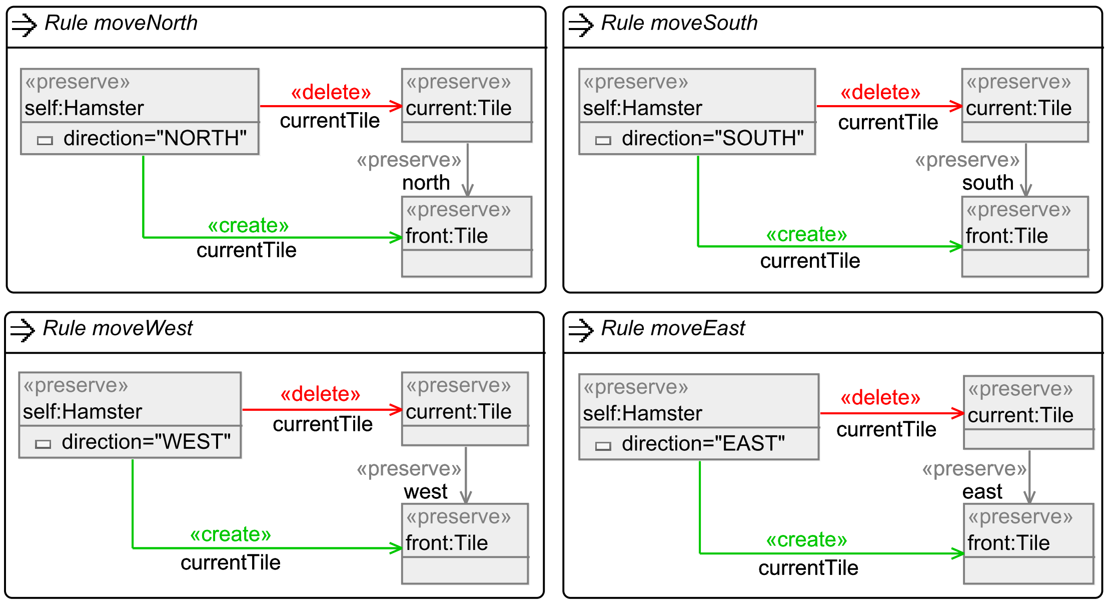

Figure 5.4: Hamster `move()` command

5.1.3 Query-DSL

For specifying queries and constraints, a custom DSL named Query-DSL is developed for this thesis. It is oriented on OCL, but omits many OCL features, which are not required for modeling a MPW. The language is developed with Xtext, which provides a practicable way to define custom DSLs.

Figure 5.5 illustrates an excerpt of the abstract syntax of the DSL, which is described in the following. The figure is based on the meta-model, which is generated by the definition of the DSL by the Xtext grammar language. The root is represented by a `Model` node, which defines the context of Query-DSL elements represented by the sub type `Context`. There are two different context variants: `CommandContext` and `ClassContext`. While the former defines the context of a command, which relates to a Henshin command modeled for the context class, the latter is used to define elements on a class level, e.g. queries or class invariants. Besides the `commandName`, a `CommandContext` also declares the parameters of a command, which can be used in the inner expressions of the element. The context declaration has to match an existing `Actor` or `Stage` class of the related MPW, otherwise errors in the later transformation phase will be thrown. In case of command contexts, additionally the method signature to a related Henshin command has to match. `Context` also defines a composition reference `elements`, which is used for the collection of `Element` objects. `Element` is used as the base type for queries and constraints. It defines an attribute `name`, which is required for queries and optional for constraints. Further, it also contains an attribute `documentation`, which contains an optional string documenting the element in natural language. These documentation values are used e.g. to generate JavaDoc comments in case of the Java code generation. Each element can consist of multiple `Expression` objects, which are used to specify the behavior of the element. The type `Expression` has multiple sub types, which are omitted in the figure and outlined in a later paragraph.

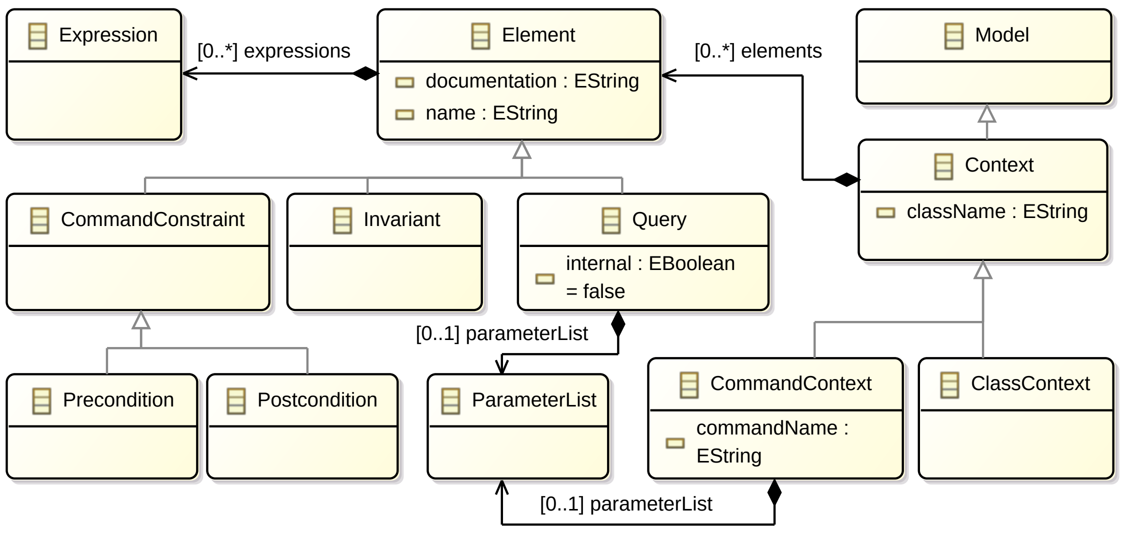

Figure 5.5: Excerpt of the Query-DSL abstract syntax meta-model

There are the following sub types of `Element`:

- `CommandConstraint`: This element represents a base type for constraints, which are defined in the context of a command. It has two derivations for pre- and postconditions.
- `Precondition`: This is the first sub type of `CommandConstraint`, which represents a precondition element for a related command. Its expressions return boolean values, which have to pass to allow an execution of the command.
- `Postcondition`: The second derivation of `CommandConstraint` is used for postconditions, which also use expressions evaluating boolean values. In contrast to preconditions, they represent rules to be valid after the execution of the related command. Additionally, `OldValueStatement` statements are allowed to be used in expressions of postconditions. These statements refer to values, which are present at the beginning of a command execution.
- `Invariant`: This element represents invariants and are defined on a class context. They contain boolean expressions, which have to be valid before and after the execution of any command related to the context class.
- `Query`: The last element type in the Query-DSL is used for queries. These elements are defined on a class context, since they are intended to extend the class by side-effect free methods. In addition to a required name, they optionally define a list of parameters, which can be used in expressions similar to command parameters. By default, queries are used to extend the facade of the related context class by a public method. To avoid this, a query can be hidden from the facade by setting the `internal` attribute to true. In contrast to constraints, `Expression` elements of a query can also return non-boolean values. Queries can also be called in constraints or other queries to allow reuse.

With a custom validator class `QueryDslValidator`, the implementation of the DSL ensures, that command contexts only contain `CommandConstraint` objects. Additionally, it ensures, that a class context can only contain `Invariant` and `Query` elements.

The essence of each `Element` is represented by its `Expression` objects, which are stored in the `expressions` containment reference like mentioned above. Most expressions are common binary operations, like `AndExpression`, `OrExpression` and `EqualityExpression` which are used to evaluate boolean values. Further, an `ImpliesExpression` is used to have a convenient syntax to express implications, e.g. for evaluating a sub expression in case of a specific direction of the hamster. There are also numeric expressions for addition, subtraction, multiplication or division of numbers. While these expressions introduced so far are used for a binary composition of sub expressions, there are atomic ones as well. On the one side, constants are expressed by `IntConstant`, `StringConstant`, `BoolConstant` or `NullConstant`. On the other side, there is a special `StatementsExpression`, which is used to store *statements*. A statement is used to express a single value by accessing parameters or variables and is represented in the abstract syntax by the `Statement` type. Further, statements can be chained in a `StatementsExpression`, which e.g. can be used to navigate through multiple properties or methods of a given variable. Besides simple statements for accessing single target values, a `Statement` object can also be bound to a `CollectionMethod`. These methods are used to call dedicated methods on a collection, e.g. for accessing the size, accessing an object at a given index or performing a type selection. A type selection works like the `typeSelect()` of Xtend, where the collection will be filtered by objects with a given type.

In the following, examples of Query-DSL models for the hamster simulator are presented to illustrate the concrete syntax. First, Listing 5.1 shows an exemplary query, which is used to check if the front of a hamster is clear. In the first line, the class context for the type `Hamster` is defined, followed by the keyword `query` to indicate, that a query will be specified. For the `frontIsClear()` query, it has to distinguished between each possible direction of the hamster. Each of the four directions is checked with an `implies` expression, which are composed by `and` expressions. In the right part of each `implies` expression, the related neighbor of the current tile is taken through statements. Further, the contents of these tiles are filtered by a `typeSelect`, to check if any object of type `Wall` is included. Above the query keyword and its name, a documentation comment is added, which includes a human readable description of the query. This documentation will be used to generate a JavaDoc comment on the `frontIsClear()` query on the Java based `Hamster` facade.

Listing 5.1 Hamster `frontIsClear()` query

```
context Hamster
/** Checks the front of the hamster. */
query frontIsClear:
        ( self.direction = WEST implies
            self.currentTile.west.contents->typeSelect(Wall)->isEmpty() )
    and ( self.direction = EAST implies
            self.currentTile.east.contents->typeSelect(Wall)->isEmpty() )
    and ( self.direction = NORTH implies
            self.currentTile.north.contents->typeSelect(Wall)->isEmpty() )
    and ( self.direction = SOUTH implies
            self.currentTile.south.contents->typeSelect(Wall)->isEmpty() );
```

The second example of a Query-DSL module is shown by Listing 5.2, which specifies a pre- and postcondition for the `pickGrain()` command. Different to the previous example, a command context is declared by extending the `Hamster` type with the name of the target command `pickGrain()`. Since no parameters are used for this command, no further parameter list has to be attached. The first constraint is a precondition, which checks if any grain is available on the hamster's tile. Because `grainAvailable()` is defined as a query in another Query-DSL module, it can be called as a statement. The second constraint is a postcondition, which checks that the number of grains in the hamster's mouth has to be increased by one. While the expression `self.grains->size()` returns the amount of grain after executing the command, the expression `old(self.grains->size())` returns the amount at the begin of the command execution. With this usage of an `old` expression, the relative effect on a specific state information can be checked. Both constraints are also enhanced by a documentation comment. These texts will be used as the exception message in the generated code, which is used to better describe the violation in case of not fulfilling the given conditions.

Listing 5.2 Hamster `pickGrain()` command constraints

```
context Hamster::pickGrain

/** there have to be grains available on the hamster's tile */
precondition: self.grainAvailable();

/** the number of grains has to be increased by one */
postcondition: self.grains->size() = old(self.grains->size()) + 1;
```

The third example shows an invariant, which is given by Listing 5.3. Like the `frontIsClear()` query, it is defined on the `Hamster` class context. The keyword `invariant` is used to mark the expressions as an invariant, followed by the optional name `isInitialized`. The invariant checks before and after each command, if the hamster is correctly initialized. This is done by the heuristic, to check if the `stage` property is set and if the hamster is placed on a tile.

Listing 5.3 Hamster `isInitialized` invariant

```
context Hamster
/** Invariant which checks if the hamster is placed on a tile. */
invariant isInitialized: self.stage <> null
                            and self.currentTile <> null;
```

To summarize, the Query-DSL is effectively used to describe queries and constraints for the modeling of the hamster simulator or other MPWs. The custom DSL has some advantages, like having full control on features which are necessary for the MPW domain. Further, it allows to give a higher usability, e.g. by using directions as built-in literals and omitting many features compared to OCL, which are not required. Currently at time of writing, the language has no intelligent auto-completion, which might be extended in the future by suggesting available properties on a given context.

5.2 Intermediate Transformation

While the previous section illustrates the input modeling, this section will outline how these input models are transformed into intermediate models, which are better suited for code generation. Figure 5.6 shows relevant artifacts and activities of the transformation phase. First, the *write activity* includes the creation of QVT-O transformations based on several meta-models. On the one side, the Ecore, Query-DSL and Henshin meta-models are used for accessing the abstract syntax trees of the input models. On the other side, the *QueryBehaviors* and *CommandBehaviors* meta-models for the intermediary models are additionally used.

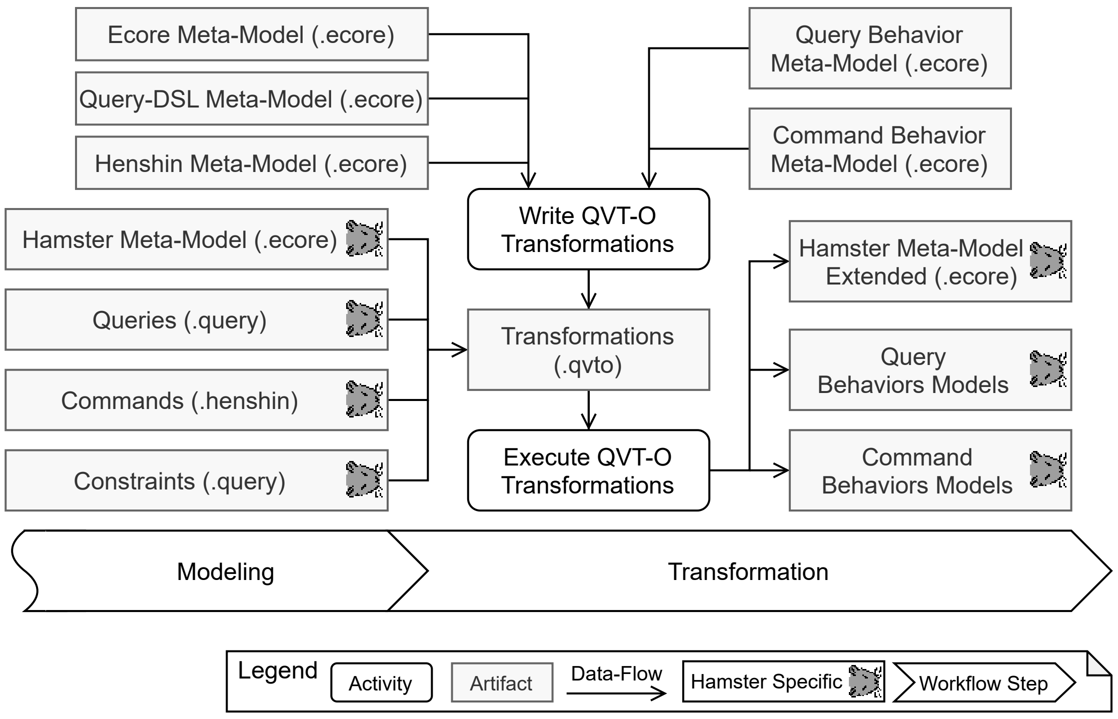
Ecore Meta-Model (.ecore)
Query-DSL Meta-Model (.ecore)
Henshin Meta-Model (.ecore)
Query Behavior Meta-Model (.ecore)
Command Behavior Meta-Model (.ecore)
Write QVT-O Transformations
Hamster Meta-Model (.ecore)
Queries (.query)
Commands (.henshin)
Constraints (.query)
Transformations (.qvto)
Execute QVT-O Transformations
Hamster Meta-Model Extended (.ecore)
Query Behaviors Models
Command Behaviors Models
Modeling
Transformation
Legend
Activity
Artifact
Data-Flow
Hamster Specific
Workflow Step

Figure 5.6: Transformation phase of the MDSD workflow

As the second activity, the QVT-O transformations are *executed*. On execution time, the four artifacts *hamster meta-model*, *queries*, *commands* and *constraints* have to be provided, which are results of the previous modeling phase. After executing the transformation phase, the hamster meta-model is extended by aspects like annotations, roles and facade classes. Additionally, the query and command behavior models are created, which contain the essence of the Henshin and Query-DSL input models. These three artifacts are the result of the transformation phase and are used as input for the generation phase, described in Section 5.3.

Figure 5.7 gives a brief overview of the intermediate models, while the following subsections will describe each meta-model in more detail. There are four packages designed for intermediate models:

- *CommandBehaviors*: Used for transformed Henshin commands, which are structured in a simplified model.
- *QueryBehaviors*: Allows to store models for queries and constraints, which are transformed by Query-DSL input models.
- *GenerationAnnotations*: Contains custom `EAnnotation` types which are used to enrich Ecore models by further information, e.g. hints how an operation body may be generated.
- *BehaviorsBaseTypes*: Provides common meta-types which are used by the three previously mentioned meta-models.

The main intent of these intermediate models is a closer focus on code generation. For example, the CommandBehaviors models have ordered references, such that the generator does not need to further consider dependencies on edges in the object graph of the transformation rules. In contrast

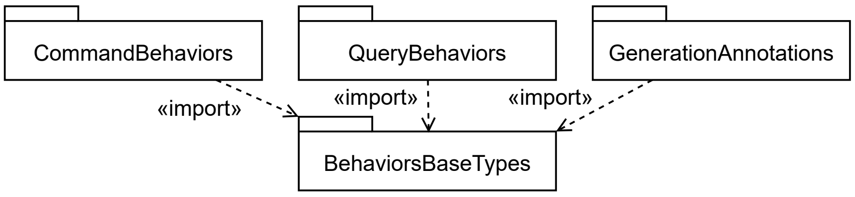

Figure 5.7: Overview of intermediate meta-models

to the Query-DSL, which is based on the concrete syntax specified by the Xtext grammar, the QueryBehaviors also focuses on generation instead of input modeling. It provides much more type information for statements and expressions, while referring to elements of the related Ecore model like `EStructuralFeature` instances. As a positive side-effect, these intermediate models decouple the generation aspects from the concrete input technology. This would allow to exchange Henshin with another technology to model commands, which only requires to adapt the related transformation scripts.

5.2.1 BehaviorBaseTypes Package

Like mentioned, the *BehaviorBaseTypes* package provides common types used in the intermediate models. Figure 5.8 shows the class diagram for this package. There are three base types defined, which are similar to Ecore types `EModelElement`, `ENamedElement` and `ETypedElement`[1]. The first one is `AnnotationableElement`, which defines a containment reference collection for `EAnnotation` elements. `NamedElement` is the second base type, which defines the attribute `name` of type `EString`. The third base type is `TypedElement`. It defines an `EClassifier` in the field `type` to reference Ecore type elements. Additionally, it specifies with `isCollection` if the type is used as a collection. With `isOptional`, the type can further be marked as an optional. Compared to `ETypedElement` of the Ecore meta-model, `isCollection` is like setting `upperBound` to `-1`, while `isOptional` relates to setting `lowerBound` to `0`. Besides the three base types, the common type `Parameter` is also defined in this package. It derives from `NamedElement` and `TypedElement` and represents parameters used in operation signatures.

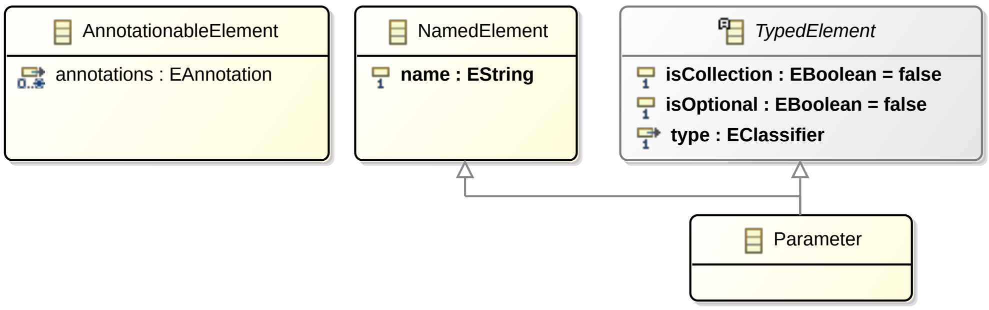
AnnotationableElement
0..* annotations : EAnnotation
NamedElement
1 name : EString
TypedElement
1 isCollection : EBoolean = false
1 isOptional : EBoolean = false
1 type : EClassifier
Parameter

Figure 5.8: BehaviorBaseTypes package

[1]Due to technical reasons, the similar Ecore types are not used as base types

5.2.2 CommandBehaviors Package

The first intermediate meta-model is realized by the *CommandBehaviors* package. Figure 5.9 shows the structural types of this package, which focuses on the *Unit* level.

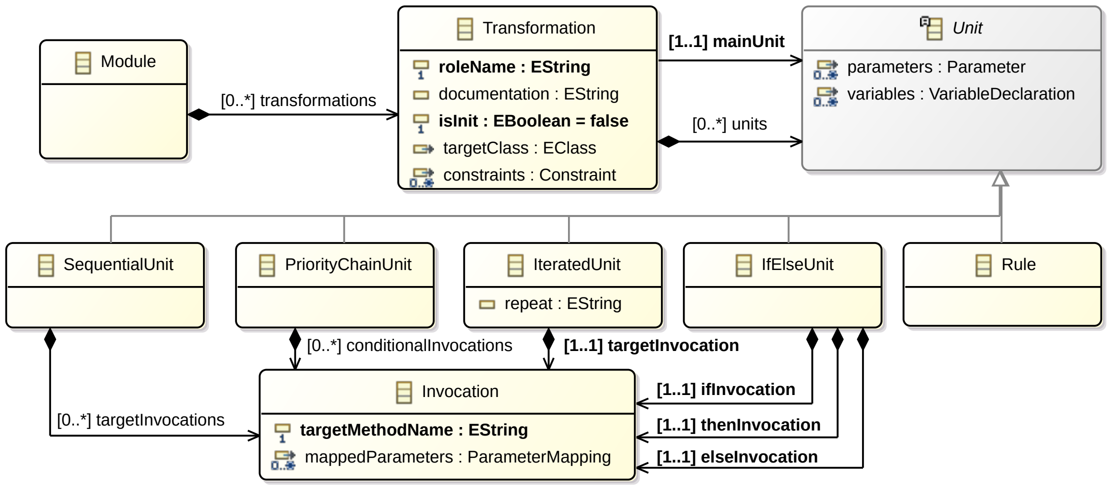

Figure 5.9: Excerpt of units in the CommandBehaviors meta-model

Module acts as the root container to store up to many Transformation objects. The type Transformation represents the central class in this package, which is related to a game or editor command. Omitted in the figure for simplicity, it derives from NamedElement and AnnotationableElement to define a name field and optional annotations. It also contains several meta-information attributes. With roleName the role is noticed, which contains either "game" or "editor". A documentation can also be defined, which is usually taken from the documentation of the related Henshin module. Further, with isInit the transformation can be marked as a command, which is used for initialization. This has to be considered when generating invariants, since conditions shall not be checked before the initialization of an entity has finished. Since each command is based on a target class, the field targetClass is defined, which references the EClass relating to an actor or a stage. Each transformation also defines up to many Constraint objects, which are part of the QueryBehaviors meta-model described below in Section 5.2.3.

Similar to Henshin commands, the modifying behavior is represented by Unit elements, which can be invoked and result either in a successful or failed state. They are stored in the units containment reference collection of Transformation, while exactly one of them has to be further set as the mainUnit of the transformation. Each unit can define up to many Parameter instances, while the main unit's parameters are used as the parameters for the whole command. In addition, Unit defines a collection of type VariableDeclaration, which is used to define variables in the unit's scope. The CommandBehaviors package defines five units, while the first four are used for structuring the control flow. They are compositions of sub units, which are invoked during runtime. An invocation is represented by the Invocation type, which defines the targetMethodName for the generated method relating to the target unit. Additionally, the collection mappedParameters specifies the parameters which will be passed as arguments to the invocation of the related method.

The control flow units are designed as following:

- `SequentialUnit`: Contains a collection of `targetInvocations`, which are invoked in sequential order. If any of the invocations is not successful, the `SequentialUnit` is marked as failed.
- `PriorityChainUnit`: Similar to the sequential one, this unit specifies a collection of invocations. But in contrast, only if the execution of a unit is not successful, the next one will be tried to be evaluated. If no invocation in `conditionalInvocations` is successful, the invocation of the `PriorityChainUnit` is handed as failed, otherwise as successful.
- `IteratedUnit`: Defines a unit, which repeats the invocation specified by `targetInvocation` a certain number of times. The attribute `repeat` contains the numeric literal or variable name which is used to evaluate the number of times to repeat. The `IteratedUnit` is only successful itself, if all iterations are completed and all target invocations are successful.
- `IfElseUnit`: This unit is used for branching the control flow. The `ifInvocation` defines the conditional unit, which is invoked to control the branch. If the invocation results as successful, the `thenInvocation` is called. Otherwise, if it results as failed, `elseInvocation` is invoked. The result of `thenInvocation` or `elseInvocation` is used to mark the `IfElseUnit` as successful or failed.

The last unit is represented by the type `Rule` and represents a graph transformation rule. In contrast to the control flow units, it does not contain sub units and describes an atomic modification of an object graph, based on a Henshin rule. Figure 5.10 depicts another excerpt of the CommandBehaviors package, with the focus on the types relevant for `Rule`. On the left side the type `Rule` is shown again, which contains several references to sub objects.

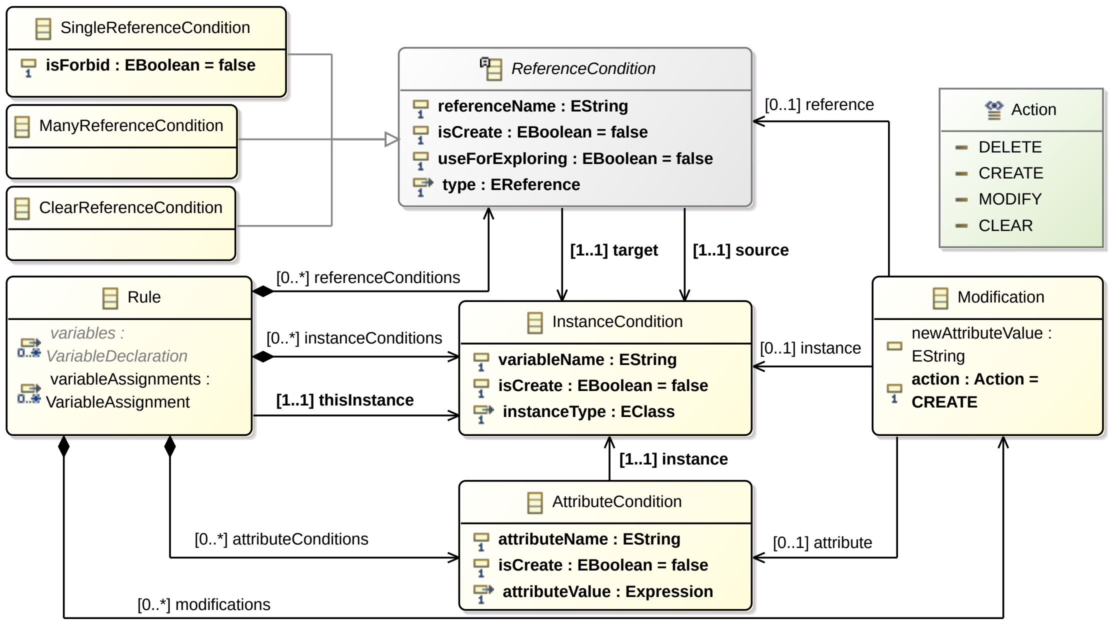

Figure 5.10: Excerpt of rule elements in the CommandBehaviors meta-model

There are three types of *conditions*, which describe the object graph to be matched:

- `InstanceCondition`: Relates to a node in the object graph and represents the condition for an object to be matched. It defines a `variableName`, which is used to store an object reference in the generated code. If the name is not specified in the Henshin input model, a unique variable name is chosen by the transformation logic. A boolean flag `isCreate` is used to indicate, if the instance is only available on the right side of the transformation rule and hence shall be created in context of this rule. Further, the type is defined with `instanceType`, which relates to an `EClass` and is determined by the type information of the related Henshin input node. `Rule` has a containment reference collection `instanceConditions` where the `InstanceCondition` objects are stored. The main instance identified by the name "self" in the Henshin rule is further referenced by the dedicated `thisInstance` reference.
- `ReferenceCondition`: Represents edges in the object graph of the transformation rule and describes conditions to be matched at runtime. Since it connects two nodes in the graph, the two `source` and `target` references are used to refer to related `InstanceCondition` objects. It defines a `referenceName` used to identify the accessor method to be called on the source instance in the generated code. Similar to the related Henshin input reference, it links to the underlying `EReference` by the `type` field. If `isCreate` is set, it defines a creating edge, which results in a setter or insertion call in the generated code. The flag `useForExploring` is evaluated for traversing the object tree to find all required `InstanceCondition` objects. If the flag is set to false, in generated code a reference will only be used for pattern matching based on previously found instances. `ReferenceCondition` itself is abstract and defines three sub types. First, `SingleReferenceCondition` relates to a simple reference, which links only to one object. As a restriction, only single references are allowed to be used with `forbid` semantics, which represents a negative application condition. The second sub type is `ManyReferenceCondition`, which is used for collection references. With `ClearReferenceCondition` a sub type is defined, which is used for clearing a collection.
- `AttributeCondition`: Defines further conditions to be matched on instance attributes. With `attributeName` the name of the `EAttribute` is noticed, while `instance` is the reference to the related `InstanceCondition` which owns the attribute. The value to be matched is stored in the `attributeValue` expression, which has to be compatible with the attribute's type. Like in previously described conditions, `isCreate` indicates if the condition is used as a modification on the right side of the object graph. If set, the attribute condition will be set on the instance. Further, the attribute value can be resolved by an `Expression` used from the QueryBehaviors package. It might refer to parameters or property paths to refer to nested properties.

While Henshin rules are designed with a left and right side graphs, for generation it is easier with a merged design like illustrated in the CommandBehaviors package. Therefore, the type `Modification` is used to link to the condition types for expressing modifications. The type defines the field `action`, which is based on the enumeration type `Action`. The enumeration consists of four actions `DELETE`, `CREATE`, `MODIFY` and `CLEAR`, which represent the according modifications. Further, `Modification` defines three links to `ReferenceCondition`, `InstanceCondition` and `AttributeCondition`, while only one of them is used at a time. In combination with the `action`, it represents a modification like deleting an instance. If the `MODIFY` action is used for an attribute, it further needs a value for `newAttributeValue`, which will be set as the new value of the attribute. Some information is explicitly redundant to make generation easier, e.g. by setting the redundant `isCreate` flag at the condition types. Further, the design is kept simple, e.g. by not deriving dedicated modification types for each consistent modification. Through validation of the input models, inconsistent states like setting multiple condition links on a `Modification` objects are prevented.

Besides conditions, the type `Rule` also contains containment references for variables. The two related types `VariableDeclaration` and `VariableAssignment` are not explicitly shown in Figure 5.10 to keep the figure more simple. First, with a `VariableDeclaration` object stored in `variables`, which is derived from the base class `Unit`, the declaration of a variable at the beginning of the generated rule's method body is inserted. It defines a `variableName` as a string and the `variableType` as an `EClassifier`. These declared variables are assigned after matching all `InstanceCondition` objects in the generated code. For assignment, an object of type `VariableAssignment` is used, which is contained in the `variableAssignments` reference collection. An assignment refers to a `VariableDeclaration` by its `variableName` and further defines an element of type `Expression` (see QueryBehaviors package below).

5.2.3 QueryBehaviors Package

This subsection outlines the next intermediate meta-model, which is about queries and constraints. It is similar to the meta-model represented by the Query-DSL language described in Section 5.1.3, which is driven by the concrete textual syntax of the Xtext grammar. Instead, the *QueryBehaviors* meta-model is targeting code generation and hence differs in several details to the related input meta-model. Figure 5.11 depicts the excerpt of the QueryBehaviors package, which focuses on the `Module` and `ExpressionalElement` types.

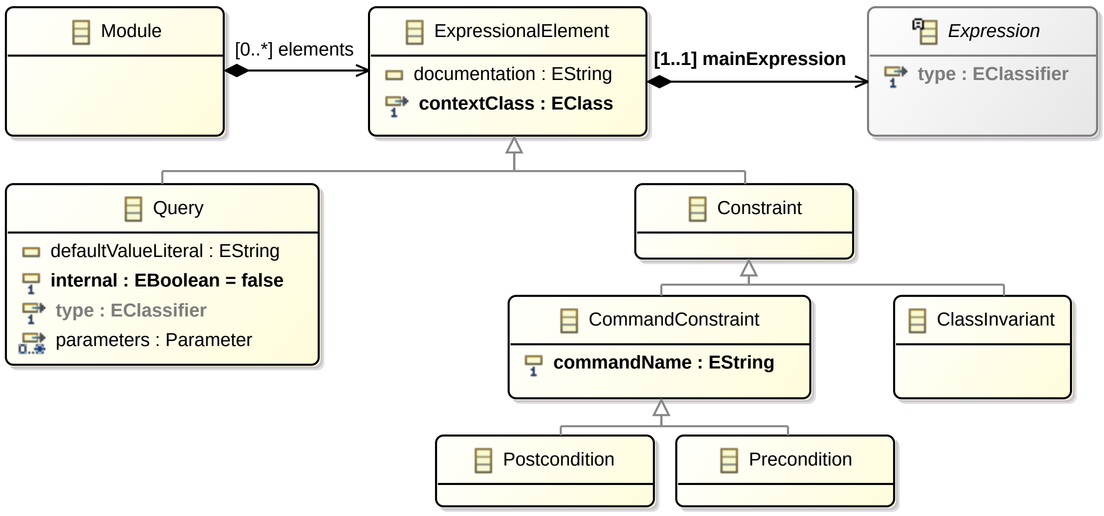

Figure 5.11: Excerpt of expressional elements in the QueryBehaviors meta-model

The type `Module` represents a root node of the model, which defines a collection containment reference `elements` of type `ExpressionalElement`. It relates to the Query-DSL `Element` type and hence is the base type for queries and constraints. Each object of type `ExpressionalElement` contains a `mainExpression` of type `Expression`, which encapsulates the operational behavior. With `Query`, `Precondition`, `Postcondition` and `ClassInvariant` the element types are represented with the same responsibility like the related types of the Query-DSL. But in QueryBehaviors, no dedicated context classes are used, therefore necessary context information is directly embedded into the related element types. For example, the parameters of queries are available in the `Query` type. As another example, the base type `CommandConstraint` simply refers to the related command by its `commandName`. If pre- or postconditions access command parameter values, the related `Expression` objects are enriched by the necessary information like the parameter name and type. In contrast to the Query-DSL, each `Expression` is enriched by a type information, therefore it derives from `TypedElement` and contains a `type` feature. Since a query also returns information of a certain type, it is derived from `TypedElement`, too. The figure does not include the `TypedElement` class to keep the diagram simple, but the inherited `type` features are visible to see the type information.

The abstract `Expression` type has the following sub types, which are used to build the operational behavior:

- `NotExpression`: Negates an inner expression, which has to be a boolean typed expression.
- `AndExpression`: Binary expression, which takes two boolean typed expressions and performs a logical *AND* on them.
- `OrExpression`: Binary expression, which takes two boolean typed expressions and performs a logical *OR* on them.
- `CalculationExpression`: Binary expression, which takes two numeric typed expressions and performs a basic calculation on it. The calculation is controlled by the enumeration type `CalculationOperator` allowing the values `PLUS`, `MINUS`, `MULTIPLY` and `DIVIDE`.
- `ImpliesExpression`: Binary expression to evaluate a logical implication. If the first, boolean typed expression is true, it will evaluate the second expression, which must also return a boolean value.
- `CompareExpression`: Binary expression, which compares two values given by sub expressions. The operator is set by the enumeration `CompareOperator`, which defines the literals `EQUAL`, `NOT_EQUAL`, `GREATER`, `SMALLER`, `GREATER_EQUAL` and `SMALLER_EQUAL`.
- `VariableExpression`: A special expression, which is intended to introduce a new variable with a given name. The value of the variable is evaluated by a sub expression. Variables introduced with a `VariableExpression` can be used by a `VariableReferenceUsageStatement`, which will be described below.
- `OldExpression`: Wraps an inner expression to indicate, that the value shall be evaluated before performing the body of the related `ExpressionalElement`. Like mentioned in the Query-DSL, through validation it is ensured, that only `PostCondition` elements can use old expressions.
- `StatementsExpression`: Represents an expression which refers to an value built by *statements*. In contrast to Query-DSL statements, there are more dedicated sub types to be better suited for generation.

In many expressions there are variables, parameters or properties of entities that are accessed with `Statement` objects. Like mentioned above, they are embedded into a `StatementsExpression` to be used within an expression tree. The type `Statement` derives from `TypedElement` and therefore defines a type, which results on evaluation. Further, it defines a `previous` and `next` reference to other `Statement` objects, to realize a chain of statements. This is used to access multiple properties in a *property path*. An example of the `grainAvailable()` query is shown in Figure 5.12, where a self statement initially accesses the `this` instance variable of the type `Hamster`. Based on this statement, it refers to a `FeatureStatement` by its `next` reference for navigating to the `currentTile` property of the `Hamster` type. Then, the next statement navigates to the feature `contents` of the `Tile` class, which is used to provide a collection of `TileContent` instances. The fourth statement on the bottom right performs a *type select* collection method, to filter out only instances of type `Grain`. Finally, the last statement calls a collection method to check, if the filtered collection is not empty. Since the last statement is typed as a boolean, the expression containing these statements also gets the boolean type. For simplicity, the figure omits the fact, that intermediary results are inserted like described below.

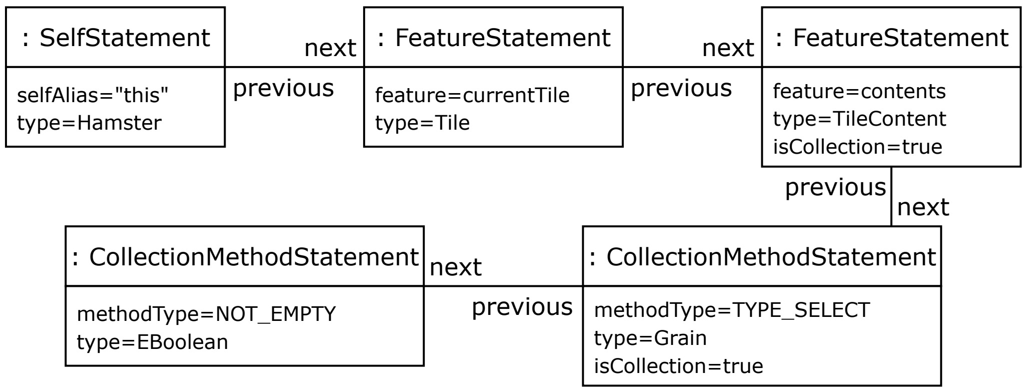

Figure 5.12: Property path example realized with statements

To build property paths for different use cases, several `Statement` sub types are defined:

- `SelfStatement`: Allows to access the *this* instance or another dedicated object, which represents the context object of the current class. With the attribute `selfAlias`, the name of the self variable is defined. A `SelfStatement` is often used as the starting point of a property path.
- `FeatureStatement`: Realizes a statement to refer to an `EStructuralFeature` of a given type. Usually, it is set as the `next` statement of a previous statement, which obtains an object of the type containing the feature. It defines an attribute `directAccessible`, which indicates, if the feature is directly accessible or a getter operation has to be used.
- `GetPropertyStatement`: A special statement which is similar to `FeatureStatement`, but does not directly relate to an `EStructuralFeature`. It is used for calling getter operations related to properties, which are not directly available in the given class. Examples are the getter operations on the role interfaces of the hamster. The name of the target getter operation is defined by the attribute `propertyGetterName`.
- `QueryStatement`: Allows to call an operation based on a query. It has a reference `query` which links to the related `Query` instance. If the query requires parameter values, the parameter names have to be specified by the attribute `parameterNames`. These parameter names are used to refer to variables or parameters of the current context.
- `IntermediaryResultStatement`: Represents a technical helper statement, which encapsulates sub statements as an intermediary result. It derives from `VariableExpression`, since it introduces a new variable by its intermediary result. The transformation logic determines

with a heuristic, which statements have to be handled by an intermediary result. The main intent is to allow a safe return in case of absent property values. Section 5.2.4 will describe the implementation of these statements in more detail.

- `ElementByNameUsageStatement`: Represents a simple statement to refer to parameters or variables by providing their names. Besides `SelfStatement`, they provide a further construct to start a property path.
- `CollectionMethodStatement`: Provides a statement, which can be applied on collections only. It defines an attribute `methodType`, which indicates which operation shall be applied by using the enumeration `CollectionMethodType`. Possible methods are `NOT_EMPTY`, `IS_EMPTY`, `SIZE`, `TYPE_SELECT` and `AT`. While `NOT_EMPTY`, `IS_EMPTY` and `SIZE` are parameter-less methods, `TYPE_SELECT` and `AT` require a parameter. Hence, the property `parameters` can be used to set parameter objects, which have to be compatible with the method type.
- `EnumLiteralStatement`: Used to refer to an enumeration literal, like directions.
- `VariableReferenceUsageStatement`: An alternative statement for accessing variables. While `ElementByNameUsageStatement` refers to a variable only by its name, this statement type makes use of a previously defined `VariableExpression`. One use case is to refer to a `VariableExpression` inserted by the transformation logic for an old value expression to store the value on start of a command. For realizing the related postcondition, afterwards the `VariableReferenceUsageStatement` is used to link to this variable. Another use case is to reuse variables introduced by `IntermediaryResultStatement` objects.

While the expressions illustrated in this subsection are primarily used for queries and constraints, they are also reused in the transformation logic to generate certain method bodies. For example, the facade is generated with methods delegating to nested objects like the `Actor`, `Stage` or `GamePerformance`. More details about the transformation logic follows in Section 5.2.4.

5.2.4 Transformation Logic

With the transformation logic written in QVT-O, the central process of the modeling workflow is performed. It contains the transformation rules to convert input models into intermediate models, which are adjusted for code generation. The attempt in this project is to make the code generation simple by extracting complexity and logic into the transformation phase. The QVT-O rules are coarsely classified into four responsibilities, which are each separately called in a MWE2 workflow. Figure 5.13 depicts the corresponding activity flow of these transformation rules.

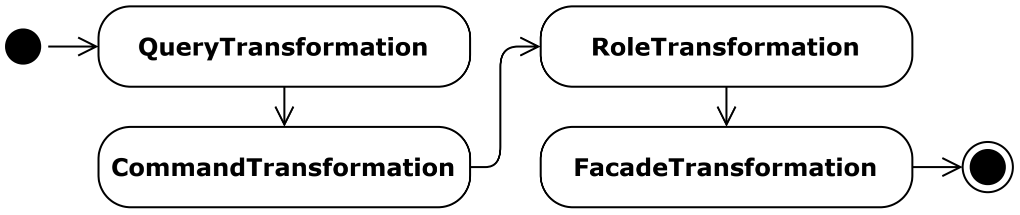

Figure 5.13: Major activities relating to rules of the transformation logic

First, the `QueryTransformation.qvto` deals with the transformation from Query-DSL modules into QueryBehaviors models. While many expressions and statements can be mapped straight forward, few types need special logic. For example, old value expressions are created by inserting a `VariableExpression` with the nested expression, which will store the evaluated value in a variable at start of the related, generated method. Then, an `OldExpression` is created with a nested `VariableReferenceUsageStatement`, which links to this `VariableExpression` instance. Further logic is implemented for intermediate results, represented by `IntermediaryResultStatement` objects. Like mentioned in context of the QueryBehaviors package in Section 5.2.3, they are used as logical wrappers around statements to introduce intermediate statement results. Their main intent is to avoid access to absent objects by returning the related expression immediately. The need for these intermediate results is determined by statements in a property path, which return optional values, e.g. by an access to structural features with optional semantics. In case any intermediate result is required for a property path, a name for a dedicated helper method is also determined. These helper methods allow to execute `return` statements in the generated helper method, if an intermediate result is empty. Since `IntermediaryResultStatement` also derives from `VariableExpression`, it can be referenced by `VariableReferenceUsageStatement` to start a new succeeding property sub path in the decomposed statements expression.

Next, the `CommandTransformation.qvto` is responsible for converting Henshin modules into CommandBehaviors models. It extracts role names from the given Uniform Resource Identifiers (URIs), transforms units and determines heuristically, which unit has to be marked as the main unit. Additionally, for actor game commands, it implicitly adds a rule to insert an entry into the game log. Finally, it iterates over the `ReferenceCondition` instances and determines the order, in which they shall be generated to have a valid graph transformation rule. To handle multiple paths leading to an `InstanceCondition`, the references on the main path are marked for exploration. Only exploring references are generated for finding new instances, while the others are only generated as checks. Further, the related constraints are embedded in the commands that were previously transformed into *QueryBehavior* elements. For actors, a dedicated write command is also added by a separate QVT-O transformation rule, while this command is not originated from a Henshin model.

The third responsibility is the transformation of roles, like introduced in Section 4.5.4. This is performed by the `RoleTransformation.qvto` file, which operates on derived types from the metatypes `Actor` and `Stage`. The original class will be renamed with the prefix ”Concrete”, since it represents the concrete implementation of the type. Then, read-only, game and editor interfaces are created as `EClass` instances with the `interface` flag set. On the concrete class, the game and editor interfaces will be set as super types, while the read-only interface becomes the super type for the game and editor interfaces. Additionally, annotations are added to these types, e.g. with a key value pair noticing the role of a class or the base name without added prefixes. Finally, the newly created role interfaces are extended by their contents. The read-only role gets query operations for the related `Query` elements and property getters for features of the concrete class. For the game and editor role interfaces, operations related to `Transformation` instances are added, which represent commands. The `Query` and `Transformation` objects are further cloned and attached to the created operations as annotation contents, to make them available at generation time.

Last, the facades described in Section 4.5.1 are inserted by the `FacadeTransformation.qvto` transformation. It creates new `EClass` objects for the game facade, which get the base names of the roles. Internally, these game facade classes contain references to the derived `MiniProgrammingWorld` instance and the concrete class, on which the facade is based on. For example, for the hamster game role, a Hamster facade class is created which references ConcreteHamster and the HamsterGame objects. Game facade classes get operations for each game command and each query defined on the related roles. These facade operations are extended by dedicated GenerationAnnotation instances, which inherit from EAnnotation to allow to add custom content to the annotation collections of EModelElement objects. The generation annotations are defined in a further meta-model named *GenerationAnnotations* and cover common low-level constructs like calling methods, defining constructors or performing field assignments. For game commands, annotations to call the commands on the game role interface are added accordingly. Further, each query is also created as an operation on the game facade, which contains generation annotations to indicate which query shall be called. For the game facade of the Actor, operations for readNumber() and readString() are also added, which delegate to the respective method on the game instance. Additionally, if any game command is marked by isInit, it will be generated as an init() facade operation used in a special constructor with matching parameters. Besides game facades, for the editor role of the derived Stage class a stage builder class is generated as well, which gets delegation operations for the editor commands in a fluent API style. Finally, the derived MiniProgrammingWorld class is also modified. For control of the game mode, it is extended by delegates for several methods of the GamePerformance, e.g. startGame() or pause(). Instead of containing the concrete class for a stage, the reference is replaced by the facade stage. For example, HamsterGame finally contains a Territory instance.

5.3 Code Generation

Following on from the transformation phase in the previous section, this section shows the generation phase based on a *model-to-text* transformation.

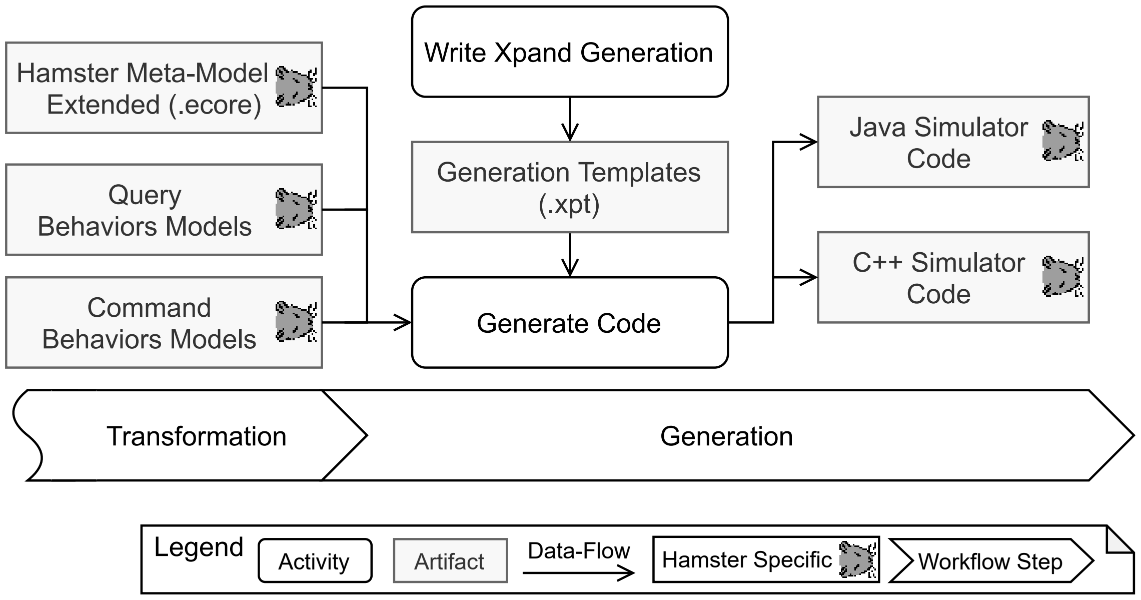

Figure 5.14: Generation phase of the MDSD workflow

Figure 5.14 depicts this phase on the MDSD workflow, which takes the transformed models as input to generate Java and C++ code. There are several Xpand templates implemented for the code generation, which are called by a MWE2 workflow file using the built-in generator, which supports to integrate Xpand. The core activity of the generation phase is the writing of these Xpand templates for the two target languages each. As resulting artifacts, the Xpand templates with the file extension `.xpt` are produced, which are used in the second activity to generate the code in conjunction of the workflow. The inputs are extended entity models, QueryBehaviors models and CommandBehaviors models. As described in the last section, entity models are extended in the transformation phase, e.g. by roles, facades or dedicated generation annotations to provide more information for the generator.

The first four subsections will go into the major responsibilities for generating code regarding entity models, queries, commands and facades. These sections will focus on the Java based generator exemplary, while the last Section 5.3.1 describes identified challenges, especially when generating C++.

5.3.1 Code Generation of Entities

The first responsibility is given by the `JavaEcoreEntitiesTemplate.xpt` template, which generates entity models. Its entry-point accepts a list of `EPackage` objects, which are the root nodes of Ecore models. Each `EClassifier` instance will be iterated and handled in one of four cases:

- `EClass` Entities: Includes types, which are no interfaces and do not derive from the `ValueType` stereotype. They are used to generate several entity classes like `MiniProgrammingWorld`, `Tile`, `GameLog` or concrete ones like `ConcreteHamster`. One major aspect for entity classes are their properties and accessor method based on structural features. Dependent on the `changeable` attribute of the `EStructuralFeature` meta-type, a property is generated as observable or final. Observable properties are generated with property wrappers, which provide observer mechanisms automatically. In case of Java, the JavaFX base properties are used, which can be used without being dependent on the JavaFX UI modules. For every property, a getter method is generated, while for observable ones a getter method to return the property wrapper is also added. Each changeable properties additionally gets a generated setter method, which is used to change the value. For `EReference` instances, which define bi-directional relations, special setter methods are generated. They also ensure to set the opposite reference. In case of collections, there are `addTo()` methods generated, which also ensure bi-directional consistency. There is further generation logic, which handles more specific cases like properties based on dependency types or changeable, but non-observable features. Besides properties, declared methods based on `EOperation` instances are also generated. On the one side, there are methods to call queries and commands, which internal structures are described in the following subsections. On the other side, usual operations are generated as abstract methods, since the implementation has to be added manually. As a consequence, the class itself is generated as abstract, while a further class with the suffix ”Impl” is generated, which is used to implement the method. This approach follows the generation pattern for *inherit and overwrite* [VSB+13].
- `EClass` Value Types: Handles types, which are derived from the `ValueType` stereotype. Examples are `Location` or `Size`, which represent simple data-structures consisting of `EAttribute` properties. For each attribute, getter and setter methods are generated. To ensure consistent equality and hashing mechanisms when using these value types, `equals()` and `hashCode()` methods are generated. Last, the generator also adds a convenient constructor method with the name `from()`, which allows to create an instance of the value type in a more readable way.

- `EClass` Interfaces: Contains interface types, which are generated to pure virtual interface classes. On the one hand, the transformed role interfaces are given by interface `EClass` instances. On the other hand, there are explicitly modeled instances like `GameViewInput` for the view model or `UserInputInterface` as a dependency interface, which is injected into the MPW model. The latter is derived from the `Dependency` stereotype, which is defined in a further Ecore package.
- `EEnum`: Includes enumeration types. In case of Java, a simple type using the `enum` keyword is generated including each defined enumeration literal.

While the generation of entity models is primarily based on static semantics, the following two subsections are dealing with generating operational behavior.

5.3.2 Code Generation of Queries

The second responsibility in context of the code generator is the generation of queries, constraints and related expressions. The related Xpand file is named `JavaQueriesTemplate.xpt`, which provides entry points for generating query operations defined on an `EClass` or to generate a given `Expression`, e.g. for command constraints. Queries are generated in the concrete classes of stages and actors. For example, for `ConcreteHamster`, all related queries are generated as methods. The signatures of these query methods are derived from the read-only interfaces, which already declare appropriate abstract operations. In concrete role classes, the bodies are generated using `EXPAND` directives for processing related templates of each `Expression` sub type. Besides expressions, the `Statement` instances are generated by Xpand templates, which are defined in a separate file named `JavaStatementsTemplate.xpt`. Helper statement methods are generated, if the transformation logic has set the `needsHelperMethod` flag on a `StatementsExpression`. These helper methods contain return statements for `IntermediaryResultStatement` instances, which are evaluated in separate variables. In the following, the generation of the `grainAvailable()` query for the hamster simulator is illustrated. While Listing 5.4 depicts the modeled query in the Query-DSL syntax, Listing 5.5 shows the generated Java code.

Listing 5.4 Hamster `grainAvailable()` query

```
context Hamster
/** Checks the hamster's current tile for grain. */
query grainAvailable: self.currentTile.contents
                           ->typeSelect(Grain)
                           ->notEmpty();
```

In the example, the expression for the query needs a statement helper method, since it navigates over the `currentTile` reference, which has an optional semantic resulting from a `lowerBound` of zero. Therefore, it has a dedicated helper method which is generated below in line 9 of Listing 5.5, which is called in the body of the query in line 4. The helper method first obtains the `currentTile` and stores it in a result variable named `result0`. Line 11 contains the presence check of the intermediate result, which returns the helper method with `false` in the absent case. Afterwards, the helper method performs the type selection statement on the `contents` reference and determines, if any `Grain` object

Listing 5.5 Generated Java code for the `grainAvailable()` query

```
1    @Override
2    public boolean grainAvailable() {
3      try {
4        return helperGrainAvailableCurrentTileResult0ContentsTypeSelectGrainNotEmpty();
5      } catch (Exception e) {
6        return false;
7      }
8    }
9    private boolean helperGrainAvailableCurrentTileResult0ContentsTypeSelectGrainNotEmpty() {
10     Tile result0 = this.getCurrentTile();
11     if (result0 == null) {
12       return false;
13     }
14     boolean result1 = result0.getContents().stream().filter(Grain.class::isInstance).map(Grain
   .class::cast)
15         .collect(Collectors.toList()).size() > 0;
16     return result1;
17   }
```

is selected. Line 16 returns the final value, which represents the full result. In case of non-boolean query types, the appropriate default value is used for returning absent property values, e.g. `null` for references.

In the next subsections, the generation of commands and facades is described, which also makes use of the generation of QueryBehaviors expressions. While commands use expressions in constraints, the facade contains dedicated `GenerationAnnotation` instances, which also allow to generate expressions.

5.3.3 Code Generation of Commands

The third responsibility of the code generator is represented by the template for commands, named `JavaCommandTemplate.xpt`. Each command is generated into a separate class, which results in a better modularization. In the concrete actor or stage classes like `ConcreteHamster`, a method for each related command is generated, which instantiates the command class and executes it with given parameters. Listing 5.6 depicts the generated example for instantiating the `move()` command of the hamster simulator.

Listing 5.6 Generated Java code for instantiating and calling `MoveCommand`

```
public class ConcreteHamster extends Actor implements GameHamster, EditorHamster { ...
  @Override
  public void move(MoveCommandParameters parameters) {
    parameters.self = this;
    var command = new MoveCommand(parameters);
    parameters.commandStack.execute(command);
  } ...
```

As shown in the listing, there is also a dedicated parameter class generated for each command, like `MoveCommandParameters`. This represents a simple data-structure, which is used to store all needed dependencies and parameters for execution of a command. The following information is stored in these parameter types:

- Self Reference: All parameter command structures are defining a `self` reference to the concrete `Actor` or `Stage` instance, which is the context object of the command. In the example, the `ConcreteHamster` object sets itself as the `self` instance.
- Command Stack Reference: Further, each command needs to be executed in context of a `CommandStack` object. The facade, which calls the command on the concrete class is responsible to set a reference to the game or editor command stack. Like illustrated in Listing 5.6, this reference is used to call the `execute()` method on the command stack.
- Game Log Reference: For every actor command, the `GameLog` instance is also set on the parameters object. This allows to insert an entry to the game log, which includes the name of the command and a reference to the `Actor` instance.
- Custom Parameters: Whenever a command defines custom parameters, these are also included in the parameters object. For example, editor commands to add props on tiles usually have custom parameters like a `Location` object.

The essence of commands is generated in the mentioned command classes like `MoveCommand`. These classes are derived from `CompositeCommandBase`, which is manually implemented in the MPW simulator framework and itself inherits from `CompositeCommand`. This base class provides helper methods, to execute primitive commands and insert them into the composite. In generated code for transformation rules, these helper methods are used to make the generated code simpler. The `execute()` method represents the entry point of each command, which evaluates constraints and executes the internal logic. Listing 5.7 shows an excerpt of the `MoveCommand` class, which includes parts of the `execute()` method.

Listing 5.7 Generated Java code for the `MoveCommand` class

```java
public class MoveCommand extends CompositeCommandBase { ...
  @Override
  public void execute() {
    if ((self.frontIsClear()) == false) {
      throw new CommandConstraintException(
        "Violation of Precondition: Hamster front must not be blocked or outside territory");
    } ...
    if (!internalMainUnit()) {
      throw new RuntimeException("Transformation was not successfully executed: move");
    }
    addGameLog();
    if ((self.getStage() != null && self.getCurrentTile() != null) == false) {
      throw new CommandConstraintException(
        "Violation of ClassInvariant: the hamster is placed on a tile.");
    }
  } ...
```

In line 4 the precondition is evaluated, which checks, if the front tile of the hamster is clear. If the precondition fails, a `CommandConstraintException` is thrown, which contains the documentation of the condition defined in the related Query-DSL file as the exception message. After precondition checks, in line 8 the `internalMainUnit()` method is called, which represents the generated unit marked with `mainUnit` in the CommandBehaviors model. Every generated unit is returning a boolean value, which indicates, if the execution has been successful. In case of a non-successful execution, a `RuntimeException` will be thrown. Usually, this should not happen, since the modeling of preconditions shall cover each invalid case to provide better semantics. After successful execution, line 11 shows that the `addGameLog()` method is called. This represents a transformation rule inserted by the transformation logic for actor commands. Finally, postconditions are evaluated. In the listing on line 12, the postcondition to check the effect of the command is performed. In case of violation, again a `CommandConstraintException` is thrown with an appropriate message. Like mentioned in the previous subsection, constraints are generated by reusing Xpand templates of the `QueryTemplate.xpt` to generate `Expression` instances into code. The next subsection will show, how commands and queries are integrated into facades, which makes interaction with MPWs more convenient.

5.3.4 Facades

While the previous subsections have depicted how the entity models and operational behavior are generated, in the following the generation of facade classes is described. The generation of facades is processed by a further Xpand template named `JavaFacadeClassesTemplate.xpt`, which primarily is based on `GenerationAnnotation` objects attached to an Ecore model. Figure 5.15 gives an overview of the different `GenerationAnnotation` types used by this generation template.

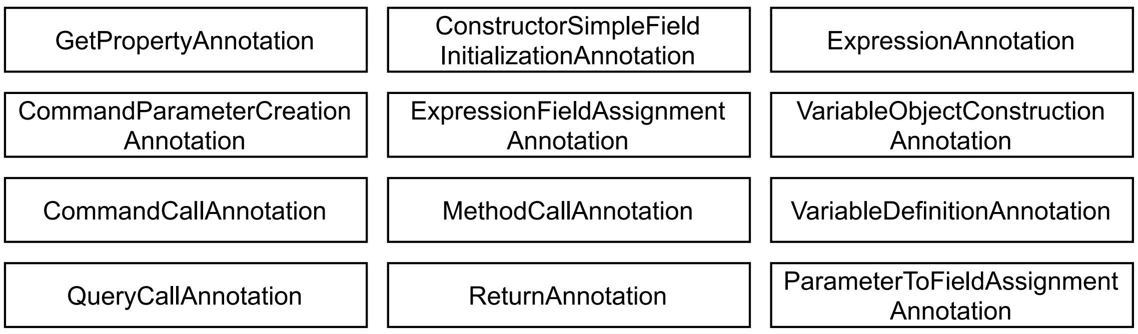
GetPropertyAnnotation
ConstructorSimpleField InitializationAnnotation
ExpressionAnnotation
CommandParameterCreation Annotation
ExpressionFieldAssignment Annotation
VariableObjectConstruction Annotation
CommandCallAnnotation
MethodCallAnnotation
VariableDefinitionAnnotation
QueryCallAnnotation
ReturnAnnotation
ParameterToFieldAssignment Annotation

Figure 5.15: Overview of `GenerationAnnotation` types

These annotation types have the following purposes:

- `GetPropertyAnnotation`: Generates a getter method, which returns a field defined on the current class. For the facade, it is used e.g. to return the internal role interfaces of the related actor or stage instance.
- `CommandParameterCreationAnnotation`: Generates code, which instantiates parameter classes like `MoveCommandParameters` and fills the required fields.
- `CommandCallAnnotation`: Used for generation of a command call in context of the current facade. It further calls appropriate methods on the `GamePerformance` method like `preExecuteGameCommand()` or `delayControlFlow()`.

- `QueryCallAnnotation`: Generates code which calls a query on the internal instance of a facade class and returns the value. For example, the `Hamster` facade has a public method for each query, which internally delegates to the `ConcreteHamster` instance.
- `ConstructorSimpleFieldInitializationAnnotation`: Defines the initialization of a field by calling a given constructor with specific parameters. As an example, the `Territory` facade class instantiates the `defaultHamster` instance in its constructor with given parameters.
- `ExpressionFieldAssignmentAnnotation`: Based on an `Expression` of the QueryBehaviors model, this annotation is used to generate a field assigment. It provides flexibility, since the expression can be used to access arbitrary property paths.
- `MethodCallAnnotation`: Generates the call of a specific method, based on a variable. For example, the `Hamster` actor facade has a `readString()` and `readInteger()` method, which internally calls the appropriate method on the `UserInputInterface` reference.
- `ReturnAnnotation`: Generates a return statement based on an `Expression`. In addition to `MethodCallAnnotation`, it is used to return the result value of the previous method call as a result of the facade method.
- `ExpressionAnnotation`: Used to simply generate an `Expression` object into code. It provides a way to insert expressions at any position of generated code. For facades, it is used to build specific parameter assignments, which are not covered by other annotations.
- `VariableObjectConstructionAnnotation`: Generates an instantiation of a given class and assigns the value to a variable.
- `VariableDefinitionAnnotation`: Defines a new variable with another `GenerationAnnotation` object, which represents the right hand side of the assignment.
- `ParameterToFieldAssignmentAnnotation`: Generates code to assign a given parameter to a field. It provides a more convenient annotation for such an assignment.

While some generation annotations are redundant to each other, they are used to explicitly make the generation as simple as possible. The facade classes are completely generated based on these annotations, which achieves that most complexity is processed in the transformation phase.

To illustrate the final control flow of a command call based on a method at the facade, Figure 5.16 shows a sequence diagram of the hamster simulator's move command. In the example, a client calls the `move()` method on an object based on the `Hamster` facade class. Since the command is defined on the game role, the generated facade method first calls `preExecuteGameCommand()` on the internal `GamePerformance`. This leads to assertion checks by the `GamePerformance` instance, e.g. to ensure that the mode is in a running state. After that, the `Hamster` class calls the `move()` method on the `GameHamster` interface, which is provided by the internal `ConcreteHamster` object. For this call, an `MoveCommandParameters` object is created with a `CommandParameterCreationAnnotation` as described above. The `GameHamster` instance then creates a `MoveCommand` object like illustrated in Section 5.3.3 and calls the `execute()` method on the `CommandStack`. After execution, the command is added to the stack collection and the control flow returns back to the `Hamster` facade class. To enforce a time delay after the game command, finally the `delayControlFlow()` method is invoked, before the command is finalized.

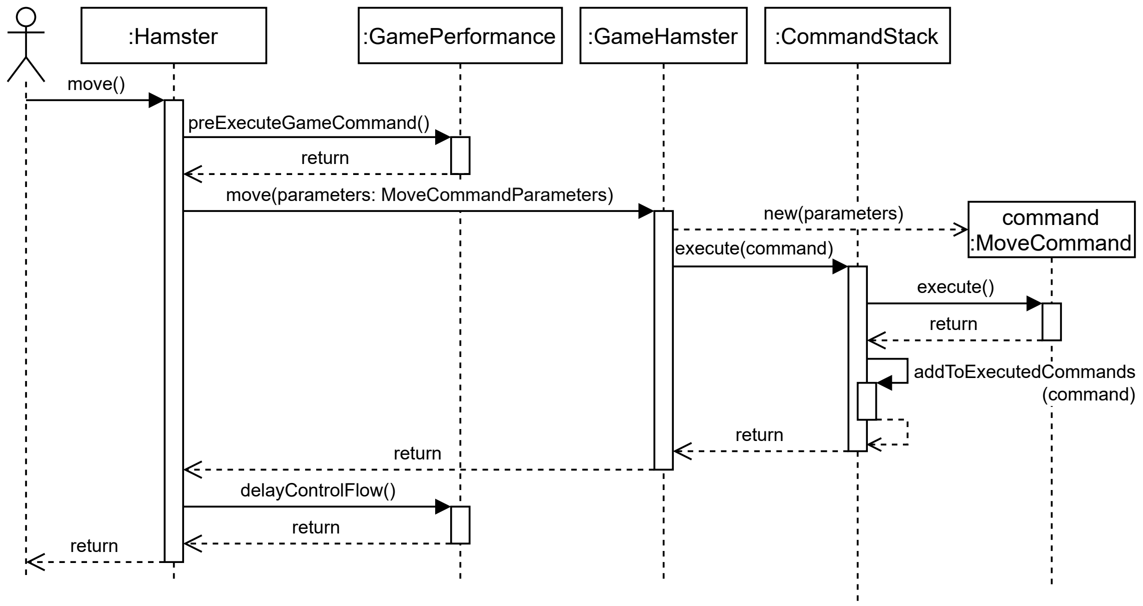

Figure 5.16: Sequence diagram for calling move command on the hamster facade

While the generation described in this subsection focused on the generation of Java, the last subsection will give a brief overview of challenges identified for the C++ generation.

5.3.5 Challenges

The generation of Java code is easier to process in several aspects compared to the generation of C++. In the following, some identified challenges are described.

First, the generated C++ entity model has to solve proper life-cycle handling. In C++, smart pointers are used to have reference counting when allocating entity instances on the heap. It is important to avoid cyclic references of shared pointers between instances, which can be achieved using weak pointers. As a distinction criteria, the `containment` attribute on `EReference` objects is used to control whether an owner semantic or only a weak reference is required.

As a second challenge the missing support of reflection has to be handled. For this, the code generator adds a lightweight mechanism to set values based on a generic value and feature keys. On the one hand, a dedicated interface is provided by the MPW simulator framework named `ReflectivePropertyObject`, which declares the three methods `setProperty()`, `addToCollection()` and `removeFromCollection()`. Each generated entity class implements these methods and handles the structural features defined on the current type. On the other hand, to express generic values, a custom `Any` type is defined, which makes use of `std::variant` of the C++ standard library to define a typed union.

A further challenge is the generation of const-correct C++ code. Especially through the usage of shared and weak pointers in collections, the generation of const-correctness for object getters is not straight forward. To overcome the problem of const-correct smart pointers, custom collection types are defined. These collection types provide a common `ObjectListView` interface, which provides iterators used to iterate over entities based on C++ references. With this design, code using collections is decoupled from life-cycle semantics, which makes the generated code more simple. Further, C++ references can be provided as const, if the collection is accessed from a const getter method. This makes generation more simple, since only the `const` keyword has to be prepended to the type.

6 Evaluation

In this chapter, the evaluation of the thesis is depicted. The work is based on a constructive approach to realize a solution with certain research related tooling. Section 6.1 first will introduce the goals of this thesis from a research perspective. Afterwards, in Section 6.2 the results are discussed.

As an overview, the following six research questions are formulated to outline the goals:

RQ1: Is it practical to develop the Proposed-Simulator by building the meta-models on the idea of the Outside-In approach?

RQ2: Which advantages and disadvantages can be achieved by the usage of the DMM technique related to modeling the hamster-simulation?

RQ3: Does the right architecture decouple the main part of the simulator from the concrete framework and allow an easy switch to other UI frameworks?

RQ4: Is it practical to use the graphical Henshin syntax to model simulator commands which are then used to generate concrete code?

RQ5: Is it practical to reuse the concepts of Fujaba to develop the basic graph transformation engine for the Proposed-Simulator?

RQ6: Is it sufficient to reuse the meta-model of Solist to build an adaptable approach which allows the creation of other MPWs?

6.1 Evaluation Goals

This section briefly states the goal of each research question in a problem-oriented manner. Each subsections sequentially relates to the questions formulated above.

6.1.1 Meta-Modeling targeting Outside-In

The first goal is to evaluate the combination of the Outside-In approach with meta-modeling. Outside-In has the focus on Design by Contract, which introduces pre- and postconditions for interfaces. Especially documentation and well-defined object-oriented design are required to teach this approach properly. Through the application of MDSD, in context of the developed Proposed-Simulator these concepts shall be applied for modeling and generated code.

6.1.2 Dynamic Meta-Modeling of MPW Simulators

The second goal also targets meta-modeling, but with focus on the dynamic aspects. DMM provides a method to define operational behavior through a runtime model and transformation rules. While the behavior of related MPW simulators is implemented mostly imperatively, the declarative modeling of behavior through graph transformation rules provides an alternative. Therefore, the second question focuses on the advantages and disadvantages of applying a DMM approach to develop MPWs. A further important goal in this context is to generate code for multiple programming languages by using the modeled semantics.

6.1.3 Architecture separating UI Views

The third goal is about the architecture of the concrete MPW simulators. To achieve that most of the simulator can be represented by similar programming language constructs, the UI framework shall be decoupled from presenter logic and core. For this, concepts like the *Humble Object pattern* suggested by the Clean Architecture are considered.

6.1.4 Command Modeling with Henshin

Next, the usage of the visual syntax of Henshin shall be evaluated in context of command modeling for MPWs. It allows the modeling of graph transformation rules in conformance to DMM in a graphical syntax. To reuse this tooling, the fourth question is about how to integrate the Henshin models into the MPW modeling workflow properly.

6.1.5 Reuse of Fujaba Ideas

The next goal is about code generation of graph transformation rules for MPW commands. Fujaba provides already ideas, how such code can be generated for Java. Therefore, the reuse of these concepts for generation of MPW commands in Java and also C++ is evaluated.

6.1.6 Reuse of Solist Meta-Model

The last evaluation goal deals with the reuse of ideas related to the Solist meta-model. This enables modeling not only specific to the hamster simulator, but also to other MPWs. As an evaluation goal, Kara the ladybug shall be modeled as an alternative MPW.

6.2 Evaluation Results

After the previous section outlined the goals for each research question, this section evaluates these aspects by reflecting the proposed solutions made in this project. Again, the subsections are sequentially ordered and relate to the research questions formulated above. Finally, threats to validity of the evaluation results are mentioned.

6.2.1 Results for Meta-Modeling targeting Outside-In

The developed solution in this thesis focuses on the Outside-In concepts. Queries and commands represent the basic building blocks to define operational behavior for MPW actors, like a hamster. Any MPW modeled by the proposed solution has a designed entity model by Ecore diagrams, which automatically focuses on object-orientation. Based on these entity models, the Henshin syntax is used to model commands, while the Query-DSL developed in context of this thesis supports modeling of queries. Further, with the Query-DSL, constraints can be modeled in addition to the transformation rules of a command. This enables a full support of Design by Contract, which is a central part of the Outside-In approach.

The generated source code for concrete MPWs retains any documentation which is added at modeling time. For example, documentation on Ecore models is generated as JavaDoc in the generated Java simulators. Pre- and postconditions are used to generate documentation on the public methods at the client facade, while they are also consistently generated as executable code. Further, one major objective in this context is to generate readable code, since the MPW simulators shall be used for teaching. Especially internals of the MPW simulators shall be available for students, since they might dive into the internal code and learn from it.

As a result, the experiences made in this thesis related to the combination of meta-modeling and the Outside-In approach are positive. Especially the definition of pre- and postconditions at modeling time allows to generate documentation and executable code from a single source, which ensures that documentation is consistent. By focusing on the relative simplicity of MPWs, the generated code can also be kept simple.

6.2.2 Results for Dynamic Meta-Modeling of MPW Simulators

Dynamic semantics for the developed MPW simulators is based on the DMM approach. The advantages and disadvantages of this approach shall be identified in this subsection.

As the central idea, the final entity model contains several dynamic references which are modified by graph transformation rules. For example, a `Hamster` object moves in the territory by updating its `currentTile` reference, which is part of the runtime model. Modifying behavior is also modeled by visual graph transformation rules, which allow a programming language independent representation. With these rules, the final source code of different programming languages is generated, which makes it possible to develop the Proposed-Simulator natively in Java or C++. It further represents a declarative modeling, which leads to simple and comprehensive transformation rules for common game commands. Another advantage is a better object-oriented design, since the runtime model enforces to have objects to be modified for operational rules. As an example, grains for the hamster simulator are modeled by a dedicated `Grain` class, such that they can be used as objects on the tiles.

As a disadvantage of the applied DMM approach, there are commands which are more complex, especially the initialization editor commands. They require multiple control flow units, combined with multiple transformation rules. While this modeling also works and is used to generated code, a procedural language might be simpler and more precise for creating a stage with its related tiles.

6.2.3 Results for Architecture separating UI Views

The architecture of the concrete MPW simulators has the goal to be independent from concrete UI frameworks. For this, the *Humble Object pattern* is applied in the proposed solution, which primarily introduces a view model and dedicated interfaces to decouple UI logic from the framework. As a resulting advantage, the manually written presenter code to fill up the view model is very symmetric across different programming languages and MPW variants. The final, UI framework dependent code has only the concern to display the contents of the view model on the screen, which does not contain business logic anymore. As another advantage, the UI framework could be exchanged more easily. Further, testing in an automated way is possible for large parts of the architecture, since unit tests can directly simulate user inputs and check visible information written in the view model. This achieves well testing capabilities.

6.2.4 Results for Command Modeling with Henshin

The fourth goal of this thesis is about the application of Henshin to model MPW commands. As result, the visual editor can be reused and no custom one has to be developed. Unlike a concrete textual syntax which is required for queries and constraints, a visual one would be more complicated to develop.

There have been two major challenges when integrating Henshin in this project. On the one hand, the modeled commands have to be integrated in the modeling workflow properly. Since the Ecore meta-model can be simply accessed by depending on the Henshin OSGi plugin, the model instances can be used in Ecore compliant tools. In this project, QVT-O is used to transform them into a proper model, which is used to contain all relevant information. On the other hand, Henshin allows to model arbitrary graph transformation rules and supports features and graph modifications not required for MPWs like the hamster simulator. To simplify the approach in this project, some restrictions have to be applied on Henshin models. This is realized by using OCL validation rules, which are evaluated after reading these Henshin models from the corresponding files.

As a final summary about the proposed application of Henshin, the integration into common Eclipse technologies like QVT-O, OCL and MWE2 works well. Further, people with Henshin experience are able to model transformations, which are used to develop commands of a MPW in a graphical and declarative representation.

6.2.5 Results for Reuse of Fujaba Ideas

In addition to the modeling of commands by Henshin, after processing a model-to-model transformation, the final target source code has to be generated. Since this generation aspect is relatively challenging, the approach of Fujaba is used as a helpful orientation. Like Fujaba, graph transformation rules are directly generated as a sequence of related object-oriented constructs. For example, the graph pattern matching is about navigating the object model from a dedicated start object marked with ”self”. Modifications call related object reference or attribute methods like setter-operations. In addition, all modifications have to be tracked in a command infrastructure, since MPWs shall support undo and redo to allow the students to analyze the runtime behavior. The extension of the Fujaba-like approach by generating primitive commands by wrappers around the modifying operations works. As a result, concepts of Fujaba are successfully applied for generating commands for the proposed MPW simulators.

6.2.6 Results for Reuse of Solist Meta-Model

The last evaluation goal deals with the reuse of the Solist meta-model to allow modeling of more than one MPW simulator. As a result, for this project the Solist meta-model is closely re-modeled as the *MiniProgrammingWorld package* introduced in Section 4.5.3. It acts as a meta-meta-model for concrete MPW meta-models like the hamster Ecore package shown in Section 5.1.1. By applying only small changes, the meta-meta-model is successfully used to model the hamster simulator, as well as Kara the ladybug. Additionally, it is also possible to use more than one `Actor` instance, which is not possible in Solist. This restriction is not given, since the approach in this project makes use of a more flexible design, where there is no need to mark a dedicated actor as a soloist instance. As an example, the hamster simulator allows to add further `Hamster` objects on the territory, to teach the object-orientation concepts like instantiating objects more properly.

6.2.7 Threats to Validity

There are some threats to the validity of the evaluation results.

First, the combination of meta-modeling with an Outside-In approach is only evaluated in the context of relatively simple MPWs. Hence, there is some threat to the *external validity*. When teaching different topics with examples not based on similar MPWs, the combination might not be useful as well. Further, the application of more complex Design by Contract concepts might not be covered by the proposed solution. For example, there might be complex constraints used for teaching, which are not supported well enough by the solution.

Regarding the DMM approach for MPW simulators, the hypotheses for a better object-oriented design and comprehensive graph transformations are based on a subjective view. This can also be seen as a threat to the *construct validity*. These statements might be invalidated through a research survey of multiple people, where e.g. students are asked with no deep knowledge of object-oriented programming or meta-modeling. Further, the adaption to other programming languages is only validated for Java and C++. As another *external validity* threat, there might be programming languages, where an adaption of the proposed solution is not practicable.

The application of the Humble Object pattern to make the UI frameworks more easily exchangeable is only based on two examples. In this case, the *external validity* is considered. The solution is designed with aspects of these frameworks in mind, which could make the integration of other frameworks more difficult. To express the degree of exchangeability in a more reliable way, the proposed solution might be researched to be rendered with several additional UI frameworks.

Commands of MPWs can be modeled with Henshin, which provides a graphical and declarative representation of transformations. The statement, that other people are also able to model MPW commands with Henshin experience, is only based on a subjective view. Therefore, the *construct* *validity* of the statement could be threatened. This works for the developers of the proposed approach with deeper knowledge, but people who have worked with Henshin in other contexts might still have difficulties with this approach.

Fujaba ideas are successfully applied for the generation of commands of MPWs. While this shall be valid for the simple MPWs used in this work, there might be MPWs not considered, where more complex commands are necessary. If these commands are too complex to be generated with the current proposed approach, this could threaten the *external validity* of the statement made in context of the evaluation result. Further, other model-driven approaches in other contexts might have different requirements, where the ideas of Fujaba are not applicable. Moreover, not all aspects of the Fujaba code generation are used. For example, the generation of entity models is not oriented on the Fujaba's approach, hence the *external validity* might only be given for simple MPWs commands.

Finally, by using the Solist meta-model, the adaptability to other MPWs is shown. But, with Kara the ladybug, this is only shown by one further example. Adaptions to many more MPWs might show, that the meta-model is not flexible enough. This would threat the *external validity* of this result. Furthermore, the adaption is made by the author of this thesis. As a further *construct validity* problem, there might be developers with less experience, which would fail to adapt other MPWs with the proposed approach.

7 Conclusion

This chapter summarizes the results and key aspects of this work. The main goal to create a hamster simulator for further programming languages based on the existing ones has been reached successfully and all important requirements in scope of this work are fulfilled.

As the essential approach, a MDSD solution is proposed, which allows to model most of the aspects of a MPW in a programming language independent way. A modeling environment is developed based on the EMF, which provides the modeling workflow for input modeling, model-to-model transformations and code generation. Entity meta-models are created by using Ecore, like the central meta-meta-model for MPWs which defines basic meta-types for actors, stages or props. Commands are modeled by reuse of the Henshin tool, which provides a visual editor to model graph transformation rules. Also, a custom Query-DSL is developed with Xtext, which is used for modeling queries and constraints. These input models are further transformed using QVT-O into intermediate models, which are adapted for code generation. As the final step in the modeling workflow, the generation of executable Java or C++ code based on Xpand templates is performed.

Besides the modeling workflow, there is the simulator environment which contains the generated code for each target programming language. While common aspects are implemented in a central MPW simulator framework for reuse, concrete simulator code is implemented in context of a concrete MPW simulator. For this, an architecture is designed which allows to decouple UI concerns from the simulator's core. While most of the code in the simulator's core is generated, UI logic has to be manually implemented in context of the MPW specific presenter. By applying the Humble Object pattern, the UI frameworks used to render the views are decoupled from the UI logic itself. This makes most of the UI logic testable in an automated manner, while code dependent on concrete UI frameworks like JavaFX or SDL are free of business logic.

As concrete MPWs, both the hamster simulator and Kara the ladybug are implemented by the proposed solution. This shows, that the chosen meta-models are flexible enough to cover other but similar MPWs. By the implementation based on two programming languages Java and C++, further the adaptability for more than one programming language is shown. Finally, the API is close to the PSE-Simulator, which is based on the Outside-In approach and has the focus on a well-documented interfaces and object-oriented design.

Future Work

While the most important requirements are fulfilled and the proposed solution can be used to develop MPW simulators for Java and C++, there might be further improvements and extensions.

As one improvement the manually written code for each target language could be produced in a more automated way. Currently, presenter logic has to be implemented for each language separately, which could be solved by using an language like ALF to describe logic in an independent manner.

It might be researched, if presenter logic could even be generated without providing MPW specific logic. Further, automated tests could also be modeled using a programming language independent way. For example, with Xtext a simple testing language could be realized, which allows to create unit tests based on the related Ecore meta-models. Similar to the code generation used in the MPW simulator core, the concrete tests then could be generated for each target language. Additionally, the loader classes like `TerritoryLoader` for deserializing hamster simulator territories are currently implemented manually. Given meta-information about the serialization, this logic might also be generated completely by a future work.

An extension could be to also develop an editor tool for creating stages. The existing stage builder classes can be used to build up stages, but currently no convenient editor is given. Based on this, a serializer must also be added, which e.g. stores stages into encoded strings.

For modeling of inputs, further improvements or extensions are possible. As an example, besides graphical modeling of commands, a textual concrete syntax with *Henshin Text* could be used as an alternative. More complex commands like initialization of stages might be written more easily in the textual syntax. For modeling of queries and constraints, the Query-DSL can be further improved. Currently, it provides no auto-completion on property paths or existing types, which could be implemented using Xtext scopes. Also, when defining the command context for constraints, a scope provider could make use of the canonical file paths to find available commands to provide auto-completion.

For teaching purposes, the tracking of the command sequence as a Labelled Transition System (LTS) could be further implemented. This enhancement can be used to show that students have solved exercises in an expected way and not by cheating. Additionally, a dedicated *game won query* could be used to decide if the current game is won. For example, in context of the hamster simulator all grain has to be picked up or put in a certain way on the territory.

Another further improvement would be to integrate the existing HTTP integration developed for the PSE-Simulator. Since the API is close to the PSE-Simulator and the visual representation is given by a view model data-structure, the serialization for HTTP could be build on top of this. This would allow to render the stages in a web-browser. Another, similar enhancement could be to develop a Jupyter notebook integration. Jupyter notebooks are documents which allow to mix executable code with documentation. They can be used to design learning environments and support activities like creating lessons, lectures, courses or assignments [BBB+19]. Modern usage scenarios regarding MPW simulators could be to demonstrate example code for exercises and provide an efficient way for the students to make notes while they are typing real code.

Finally, the solution of this work could be adapted for more programming languages and MPWs. Constraints in the generated Java code could also be generated using the JML, which provides a formalized syntax. For this, the Java generator has to be improved by correctly generating rich JML documentations above method definitions.

Bibliography

[ABJ+10] T. Arendt, E. Biermann, S. Jurack, C. Krause, G. Taentzer. “Henshin: advanced concepts and tools for in-place EMF model transformations”. In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2010, pp. 121–135 (cit. on p. 17).

[AEH+99] M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schürr, G. Taentzer. “Graph transformation for specification and programming”. In: *Science of Computer programming* 34.1 (1999), pp. 1–54 (cit. on pp. 12, 13).

[ALS16] A. Anjorin, E. Leblebici, A. Schürr. “20 years of triple graph grammars: A roadmap for future research”. In: *Electronic Communications of the EASST* 73 (2016) (cit. on p. 13).

[BA20] D. S. Batory, N. Altoyan. “Aocl: A Pure-Java Constraint and Transformation Language for MDE.” In: *MODELSWARD*. 2020, pp. 319–327 (cit. on p. 25).

[BB14] D. Boles, C. Boles. *Objektorientierte Programmierung spielend gelernt mit dem Java-Hamster-Modell*. Springer, 2014 (cit. on pp. 1, 20, 21).

[BBB+19] L. A. Barba, L. J. Barker, D. Blank, J. Brown, A. Downey, T. George, L. Heagy, K. Mandli, J. K. Moore, D. Lippert, et al. *Teaching and Learning with Jupyter*. 2019 (cit. on p. 82).

[BBF20] S. Becker, C. Bescherer, A. Fest. “Reflective Pedagogical Practice on and in Introduction to Programming and Software Engineering”. In: *practice* 1 (2020), p. 2 (cit. on pp. 1, 7, 20).

[BMW+17] F. Bedini, R. Maschotta, A. Wichmann, S. Jäger, A. Zimmermann. “A Model-Driven fUML Execution Engine for C++.” In: *MODELSWARD*. 2017, pp. 443–450 (cit. on p. 26).

[Bol] D. Boles. “Solist–eine Entwicklungsumgebung für Miniprogrammierwelt-Simulatoren”. In: *Informatik und Kultur* (), p. 88 (cit. on pp. 2, 20, 21).

[BSE10] N. Bandener, C. Soltenborn, G. Engels. “Extending DMM behavior specifications for visual execution and debugging”. In: *International Conference on Software Language Engineering*. Springer. 2010, pp. 357–376 (cit. on p. 10).

[CG12] J. Cabot, M. Gogolla. “Object constraint language (OCL): a definitive guide”. In: *International School on Formal Methods for the Design of Computer, Communication and Software Systems*. Springer. 2012, pp. 58–90 (cit. on p. 16).

[CH06] K. Czarnecki, S. Helsen. “Feature-based survey of model transformation approaches”. In: *IBM systems journal* 45.3 (2006), pp. 621–645 (cit. on pp. 11, 12).

[EEK+12] S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, R. von Massow, W. Hasselbring, M. Hanus. “Xbase: implementing domain-specific languages for Java”. In: *ACM SIGPLAN Notices* 48.3 (2012), pp. 112–121 (cit. on p. 16).

[EHHS00] G. Engels, J. H. Hausmann, R. Heckel, S. Sauer. “Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML”. In: *International Conference on the Unified Modeling Language*. Springer. 2000, pp. 323–337 (cit. on p. 10).

[EMF4CPP20] Senac, Andrés and Sevilla, Diego. *EMF4CPP*. 2020. URL: `https://github.com/catedrasaes-umu/emf4cpp` (cit. on p. 23).

[ESW07] G. Engels, C. Soltenborn, H. Wehrheim. “Analysis of UML activities using dynamic meta modeling”. In: *International Conference on Formal Methods for Open Object-Based Distributed Systems*. Springer. 2007, pp. 76–90 (cit. on p. 10).

[EV06] S. Efftinge, M. Völter. “oAW xText: A framework for textual DSLs”. In: *Workshop on Modeling Symposium at Eclipse Summit*. Vol. 32. 118. 2006 (cit. on p. 16).

[Fou] E. Foundation (cit. on pp. 14, 15).

[Fow04] M. Fowler. *Inversion of control containers and the dependency injection pattern (2004)*. 2004. URL: `https://martinfowler.com/articles/injection.html` (cit. on p. 38).

[Fow20] M. Fowler. *HumbleObject (2020)*. 2020. URL: `https://martinfowler.com/bliki/HumbleObject.html` (cit. on pp. 23, 31).

[Gam95] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995 (cit. on pp. 14, 31, 33, 34).

[GS07] M. Garcia, A. J. Shidqie. “OCL compiler for EMF”. In: *Eclipse Modeling Symposium at Eclipse Summit Europe*. 2007, pp. 1–10 (cit. on p. 25).

[GTC+15] S. Guermazi, J. Tatibouet, A. Cuccuru, S. Dhouib, S. Gérard, E. Seidewitz. “Executable modeling with fuml and alf in papyrus: Tooling and experiments”. In: *strategies* 11 (2015), p. 12 (cit. on p. 26).

[GZ06] L. Geiger, A. Zündorf. “TOOL modeling with Fujaba”. In: *Electronic Notes in Theoretical Computer Science* 148.1 (2006), pp. 173–186 (cit. on p. 24).

[HamsterModell20] Boles, Dietrich. *Java-Hamster-Modell*. 2020. URL: `https://www.java-hamster-modell.de` (cit. on p. 21).

[JMJ+16] S. Jäger, R. Maschotta, T. Jungebloud, A. Wichmann, A. Zimmermann. “An EMF-like UML generator for C++”. In: *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. IEEE. 2016, pp. 309–316 (cit. on p. 23).

[Küh06] T. Kühne. “Matters of (meta-) modeling”. In: *Software & Systems Modeling* 5.4 (2006), pp. 369–385 (cit. on p. 8).

[Mar18] R. C. Martin. *Clean architecture: a craftsman’s guide to software structure and design*. Prentice Hall, 2018 (cit. on p. 32).

[Mey09] B. Meyer. “Touch of Class: Learning to Program Well with Objects and Contracts”. In: (2009) (cit. on pp. 1, 5, 6, 8).

[NNZ00] U. Nickel, J. Niere, A. Zündorf. “The FUJABA environment”. In: *Proceedings of the 22nd international conference on Software engineering*. 2000, pp. 742–745 (cit. on pp. 24, 25).

[Nol10] S. Nolte. *QVT - Operational Mappings*. Springer-Verlag, 2010 (cit. on p. 18).

[Obj11] Object Management Group (OMG). *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1*. OMG Document Number formal/2011-01-01 (`https://www.omg.org/spec/QVT/1.1`). 2011 (cit. on p. 17).

[Obj13a] Object Management Group (OMG). *Action Language for Foundational UML, Version 1.0.1*. OMG Document Number formal/2013-09-01 (`https://www.omg.org/spec/ALF/1.0.1`). 2013 (cit. on p. 26).

[Obj13b] Object Management Group (OMG). *Semantics of a Foundational Subset for Executable UML Models, Version 1.1*. OMG Document Number formal/2013-08-06 (`https://www.omg.org/spec/FUML/1.1`). 2013 (cit. on p. 26).

[Obj14] Object Management Group (OMG). *Object Constraint Language, Version 2.4*. OMG Document Number formal/2014-02-03 (`https://www.omg.org/spec/OCL/2.4`). 2014 (cit. on p. 16).

[Obj15] Object Management Group (OMG). *XML Metadata Interchange (XMI) Specification, Version 2.5.1*. OMG Document Number formal/2015-06-07 (`https://www.omg.org/spec/XMI/2.5.1`). 2015 (cit. on p. 15).

[Obj17] Object Management Group (OMG). *OMG Unified Modeling Language, Version 2.5.1*. OMG Document Number formal/2017-12-05 (`https://www.omg.org/spec/UML/2.5.1`). 2017 (cit. on p. 9).

[Obj19] Object Management Group (OMG). *Meta-Object Facility (MOF) Core Specification, Version 2.5.1*. OMG Document Number formal/2019-10-01 (`https://www.omg.org/spec/MOF/2.5.1`). 2019 (cit. on p. 10).

[PM06] M. Pedroni, B. Meyer. “The inverted curriculum in practice”. In: *Proceedings of the 37th SIGCSE technical symposium on Computer science education*. 2006, pp. 481–485 (cit. on pp. 1, 6, 7).

[Roz97] G. Rozenberg. *Handbook of graph grammars and computing by graph transformation*. Vol. 1. World scientific, 1997 (cit. on p. 13).

[SB19] J. Schröpfer, T. Buchmann. “Integrating UML and ALF: An Approach to Overcome the Code Generation Dilemma in Model-Driven Software Engineering”. In: *International Conference on Model-Driven Engineering and Software Development*. Springer. 2019, pp. 1–26 (cit. on p. 26).

[SBG+17] D. Strüber, K. Born, K. D. Gill, R. Groner, T. Kehrer, M. Ohrndorf, M. Tichy. “Henshin: A usability-focused framework for emf model transformation development”. In: *International Conference on Graph Transformation*. Springer. 2017, pp. 196–208 (cit. on p. 17).

[SBMP08] D. Steinberg, F. Budinsky, E. Merks, M. Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008 (cit. on pp. 10, 14, 15).

[Sch13] A. Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen: Formale Definitionen, Anwendungsbeispiele und Werkzeugunterstützung Herausgegeben und eingeleitet von Manfred Nagl*. Springer-Verlag, 2013 (cit. on p. 13).

[Sch94] A. Schürr. "Specification of graph translators with triple graph grammars". In: *International Workshop on Graph-Theoretic Concepts in Computer Science*. Springer. 1994, pp. 151–163 (cit. on p. 13).

[SE10] C. Soltenborn, G. Engels. "Towards Generalizing Visual Process Patterns". In: *Electronic Communications of the EASST* 25 (2010) (cit. on p. 10).

[SJGE18] S. Schwichtenberg, I. Jovanovikj, C. Gerth, G. Engels. "CrossEcore: an extendible framework to use ecore and OCL across platforms". In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 2018, pp. 292–293 (cit. on p. 23).

[SMGG07] A. J. Shidqie, R. Möller, D. Gollmann, M. S. M. Garcia. "Compilation of OCL into Java for the Eclipse OCL Implementation". PhD thesis. Citeseer, 2007 (cit. on p. 25).

[Sta73] H. Stachowiak. *Allgemeine modelltheorie*. Springer, 1973 (cit. on p. 8).

[VSB+13] M. Völter, T. Stahl, J. Bettin, A. Haase, S. Helsen. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2013 (cit. on pp. 8–10, 67).

[Win15] S. Winetzhammer. "Modellgetriebene Entwicklung mit Graphtransformationen". PhD thesis. 2015 (cit. on pp. 13, 17, 24).

[Zün01] A. Zündorf. "Rigorous object oriented software development". PhD thesis. Habilitation Thesis, University of Paderborn, 2001 (cit. on pp. 24, 25).

[Zün19] A. Zündorf. "The Fulib Solution to the TTC 2019 Truth Table to Binary Decision Diagram Case." In: *TTC@ STAF*. 2019, pp. 15–19 (cit. on p. 25).

All links were last followed on April 13, 2021.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature