FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Improving the Comprehension of Domain-Specific Languages by Utilizing Visualizations

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Alexander Altenhuber
Matrikelnummer 1125773

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: O.Univ.Prof. Dipl.-Ing. Mag. Dr.techn. Gerti Kappel
Mitwirkung: Univ.Ass. Dipl.-Ing. Dr.rer.soc.oec. BSc Tanja Mayerhofer
Univ.Ass. Dipl.-Ing. Dr.techn. Philip Langer

Wien, 20.11.2016

(Unterschrift Verfasser) (Unterschrift Betreuung)

Improving the Comprehension of Domain-Specific Languages by Utilizing Visualizations

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Alexander Altenhuber
Registration Number 1125773

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: O.Univ.Prof. Dipl.-Ing. Mag. Dr.techn. Gerti Kappel
Assistance: Univ.Ass. Dipl.-Ing. Dr.rer.soc.oec. BSc Tanja Mayerhofer
Univ.Ass. Dipl.-Ing. Dr.techn. Philip Langer

Vienna, 20.11.2016

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Alexander Altenhuber
Beethovenstraße 16/14, 4020 Linz

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

First and foremost, I would like to thank my parents, Andrea and Gerald, without whom my studies would not have been possible. I am grateful for their financial, motivational, and inspirational support.

Moreover, I owe special thanks to Barbara for constantly supporting me over the last five years, showing understanding during work-intensive periods, and celebrating every single achievement with me.

I would also like to thank my fellow students and friends for countless hours of puzzling over assignments, wonderful discussions, and motivational burger dinners.

Last but not least, this diploma thesis would not have been possible without my advisors Prof. Dr. Gerti Kappel and Dr. Tanja Mayerhofer, who sparked my interest in Model Engineering throughout my Master's studies. I would especially like to thank Philip for the constant support, invaluable feedback, and creative suggestions.

Abstract

Domain-specific languages (DSLs) are popular for many reasons such as increasing productivity for developers and improving communication with domain experts. DSLs can be divided into textual and graphical DSLs. Textual DSLs let users create models by using a textual concrete syntax (TCS). Graphical DSLs render the model by means of a graphical concrete syntax (GCS), which is based on graphical shapes and also facilitates graphical editing. Using a TCS may lead to higher productivity due to editor functionalities like search and replace, syntax highlighting, and code completion. Textual models, however, may be hard to understand for novice developers and domain experts in particular. Graphical representations of models, such as GCSs, aim at improving the comprehension of models and the communication with domain experts. A GCS, however, mostly visualizes structural aspects of the model by defining a mapping between semantic elements and graphical elements. Furthermore, graphical editing capabilities may impose restrictions on the design of a GCS. Therefore, a GCS might not be the best option when solely aiming at improving the comprehension.

This thesis analyzes a way of combining both representations by using a textual DSL for editing purposes and read-only graphical representations which entirely aim at improving the comprehension of the DSL. This allows developers to fully concentrate on building graphical representations which highlight specific aspects of models and help users to better understand or interpret them. These graphical representations are referred to as *visualizations* in this thesis. A visualization is a graphical representation that cannot be edited and highlights a particular aspect. This thesis mainly aims at investigating if visualizations can increase users' comprehension of models. Furthermore, it intends to evaluate the feasibility of using JavaFX as a base technology for creating visualizations. The results of this work are evaluated based on two use cases. The first use case aims at exploring and illustrating the technical capabilities of using JavaFX as a technology for creating visualizations. The second use case intends to evaluate the practical relevance of visualizations in the domain of automotive testing by creating visualizations for an existing DSL. The answers to our research questions are based on the results of in-depth interviews, which have been conducted with engineers professionally using the DSL.

Kurzfassung

Domänen-spezifische Sprachen (DSS) zielen darauf ab, die Produktivität von Entwicklern zu verbessern und die Kommunikation mit Domänenexperten zu vereinfachen. DSS können in textuelle und grafische DSS unterteilt werden. Bei textuellen DSS werden Modelle mit Hilfe einer konkreten textuellen Syntax (KTS) formuliert. Grafische DSS zeigen das Modell mit Hilfe einer konkreten grafischen Syntax (KGS), welche auf grafischen Formen basiert und auch ein grafisches Editieren ermöglicht. Die Verwendung einer KTS kann die Produktivität durch Editorfunktionen wie Suchen und Ersetzen, Syntaxhervorhebung und Codevervollständigung verbessern. Textuelle Modelle können aber den Nachteil haben für neue Entwickler und speziell für Domänenexperten schwer verständlich zu sein. Grafische Repräsentationen von Modellen, z.B. in Form einer KGS, beabsichtigen das Verständnis von Modellen sowie die Kommunikation mit Domänenexperten zu verbessern. Eine KGS visualisiert aber vor allem strukturelle Aspekte des Modells indem eine Zuordnung zwischen semantischen Elementen und grafischen Elementen definiert wird. Weiters können Funktionen zum grafischen Editieren von Modellen das Design einer KGS einschränken. Eine KGS ist deshalb möglicherweise nicht die beste Möglichkeit, wenn nur die Verbesserung des Verständnisses angestrebt wird.

Diese Arbeit untersucht einen Ansatz der beide Repräsentationen kombiniert und eine textuelle DSS für das Editieren des Modells, sowie nicht-editierbare grafische Repräsentationen, welche nur das Ziel haben das Verständnis der DSS zu verbessern, verwendet. Entwickler können sich dadurch ausschließlich darauf konzentrieren, grafische Repräsentationen zu entwerfen, welche einen spezifischen Aspekt eines Modells hervorheben und es dem Benutzer dadurch vereinfachen das Modell interpretieren zu können. Diese grafischen Repräsentationen werden in dieser Arbeit als *Visualisierungen* bezeichnet. Eine Visualisierung ist eine nicht editierbare, grafische Repräsentation, welche einen bestimmten Aspekt eines Modells hervorhebt. Diese Arbeit erforscht hauptsächlich, ob Visualisierungen das Verständnis von Benutzern für Modelle verbessern können. Weiters soll untersucht werden, ob JavaFX eine praktikable Technologie zur Entwicklung von Visualisierungen darstellt. Die Ergebnisse dieser Arbeit basieren auf zwei Anwendungsfällen. Der erste Anwendungsfall beabsichtigt die technischen Möglichkeiten von JavaFX bei der Entwicklung von Visualisierungen auszuloten. Der zweite Anwendungsfall soll die praktische Relevanz von Visualisierungen in der Domäne des Testens von Geräten im Automobilbereich analysieren. Dabei werden Visualisierungen für eine existierende DSS erstellt. Die Forschungsfragen dieser Arbeit werden anhand der Ergebnisse mehrerer Interviews beantwortet, welche mit Ingenieuren, die die DSS regelmäßig verwenden, durchgeführt werden.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Domain-specific languages (DSLs) are popular for many reasons such as increasing productivity for developers and improving communication with domain experts [11]. DSLs consist of a concrete syntax and a metamodel. The metamodel defines language concepts and how these concepts can be combined. The concrete syntax, which is mostly textual or graphical, defines the notation with which users can express programs. A DSL can have both graphical and textual concrete syntaxes. Using a textual concrete syntax may lead to higher productivity due to text editor functionalities like copying, pasting, searching, replacing, syntax highlighting, and code completion. However, compared to graphical representations, textual programs are often harder to understand for novice developers and for domain experts in particular. The goal of graphical concrete syntaxes and software visualization in general is to produce "computer images which evoke mental images for comprehending software better" [10]. Therefore, using both textual syntaxes for increasing productivity and graphical syntaxes for improving comprehension and communication clearly facilitates the main strengths of DSLs which have been mentioned above.

1.2 Problem Statement

Software visualization in the area of DSLs is mostly realized by designing graphical concrete syntaxes for textual languages. Graphical representations are usually used for improving comprehension and are most often designed in a way that allows editing of the program in a graphical form as well. Therefore, graphical languages mostly visualize structural aspects of the program by defining a mapping between semantic elements and graphical elements. Graphical editing capabilities may impose restrictions on the design of a graphical concrete syntax which might weaken the graphical language's power to improve comprehension. For example, a DSL's users may require a complex three-dimensional model visualization in order to improve communication with domain experts. Developers may be intimidated by the complexity of facilitating graphical editing support for a three-dimensional representation and, therefore, choose to implement a two-dimensional representation instead. However, this graphical representation may not sufficiently address the users' needs.

This thesis analyzes a way of designing graphical representations which entirely aim at improving comprehension of DSLs by dropping the need of graphical editing. This allows designers to fully concentrate on building graphical representations which highlight specific aspects of a program and help users to better understand or interpret them rather than worrying about facilitating the editing process. Based on the definition by Voelter et al. [30] these graphical representations are referred to as *visualizations* in this thesis. "A visualization is a graphical representation of a model that cannot be edited. It is created from the core model using some kind of transformation, and highlights a particular aspect of the source program." [30] A visualization can also consider additional semantics and can include structural, behavioral, and evolutional information. In case of a circuit diagram, a visualization could, for example, show which elements of the circuit are not energized considering a given input voltage. Another visualization of textual information can be found in most LaTex editors in form of a PDF preview. In a language similar to state-machines a visualization could check for and show impasses.

Developers and users have high expectations of a technology used for realizing visualizations. Users expect highly customized and modern graphics, suggestive metaphors, and a strong emphasis on usability. It should be possible to only view areas of interest and discover other parts of the visualization if necessary. Dynamic visualizations (e.g. used for visualizations of program executions) often require advanced animations. Sophisticated graphical representations are hardly useful if users cannot easily access them. Heavy-weight Integrated Development Environments (IDEs) for using DSLs are a major barrier for domain experts and should make way for browser-based editors and graphical views.

For these requirements current graphical editor frameworks like Sirius[1], Graphiti[2], or GMF[3] are limited. The main reason for this limitation is the fact that all of these frameworks are based on GEF3[4]. Since the release of GEF3 in 2004 no breaking API changes have been introduced due to multiple important commercial stakeholders. This long term backward compatibility prevented framework developers from applying necessary major API refactorings. The dependency of GEF3 on Draw2D[5] results in outdated graphical visualizations. Draw2D is based on integer coordinates which often leads to rendering issues. Moreover, it does not support complex geometry such as Bézier curves. Furthermore, GEF3 lacks support for advanced transformations, rotations, and multi-touch gestures.

[1]`eclipse.org/sirius`
[2]`eclipse.org/graphiti`
[3]`eclipse.org/modeling/gmp`
[4]`eclipse.org/gef`
[5]`eclipse.org/gef/draw2d`

JavaFX[6] provides superior graphics and animations, allows to focus on usability, leverages the hardware, and, therefore, is a highly promising technology for implementing visualizations. It is used in the practical part of this thesis.

1.3 Aim of the Work

This thesis aims at investigating the potential of using visualizations for fostering the comprehension of DSLs. The results of this work are evaluated based on two use cases. The first use case is a fictional one and aims at exploring and illustrating the technical capabilities of JavaFX for implementing visualizations. In the course of this use case a DSL for designing game levels is developed using Xtext[7]. Based on the DSL, visualizations highlighting various aspects of the model are created using JavaFX. In the second use case it is intended to evaluate the practical relevance of visualizations in the domain of automotive testing. Using an existing DSL of an Austrian automotive supplier, again graphical visualizations using JavaFX are constructed. The created visualizations are evaluated by performing interviews with real domain experts from the automotive domain.

From the requirements given in the previous section, the following research questions (RQ) can be derived:

RQ1. Do visualizations increase users' comprehension of models?
RQ2. Are visualizations superior to reports which highlight a particular aspect in textual notation?
RQ3. Is JavaFX a feasible technology for implementing visualizations for DSLs?

The main questions (RQ1 and RQ2) are concerned with answering how visualizations can be designed and if visualizations actually foster comprehension of DSLs. RQ2 analyzes whether comprehension is improved in consequence of using graphical views or due to the information obtained from interpreting a model. This question is answered by comparing visualizations, which graphically represent information gathered from interpreting a model, to reports, which sum up this information in textual form. As using JavaFX as a base technology for creating diagram views has not yet been discussed in scientific literature RQ3 aims at summing up the experiences gained with JavaFX during this thesis' practical part.

1.4 Methodological Approach

As the main result of this work is the creation of new software artifacts, i.e. graphical visualizations, the methodological approach used is design science [13]. Following the guidelines and principles of this methodology, the two main steps are *design* and *evaluation*. The first step contains the "creation of an innovative purposeful artifact for a specified problem domain" [13]. This includes the implementations of the graphical visualizations of the previously described

[6]`docs.oracle.com/javafx`
[7]`eclipse.org/Xtext`

use cases (fictional and practical). The second step should demonstrate the utility, quality, and efficacy of the developed artifacts via well-executed evaluation methods. In particular, the evaluation should yield answers for RQ1 and RQ2. To evaluate the practical use, we develop visualizations for a real-world DSL that is used by engineers in automotive testing and conducted in-depth interviews with users to learn about the visualizations' impact on users' comprehension.

1.5 Structure of the Work

This thesis consists of eight further chapters and a concluding summary in Chapter 10. Chapter 2 is concerned with domain-specific languages in general. Approaches for visualizing DSLs and a definition of visualizations and reports are outlined in Chapter 3. Chapter 4 presents the first use case and the implemented visualizations. Chapter 5 is concerned with existing frameworks for implementing visualizations. Chapter 6 discusses visualizations based on JavaFX and introduces a technology-independent architecture of visualizations. Chapter 7 reveals the detailed experiences that were gained during the implementation of the visualizations using JavaFX. The results of the evaluation are described in Chapter 8. Related work is presented in Chapter 9.

CHAPTER 2

Domain-specific languages (DSLs)

This chapter aims at providing an overview of the most important characteristics of domain-specific languages (DSLs). The core properties of DSLs - especially goals, the concrete syntax, the metamodel, and the semantic model - are also important for creating effective visualizations and will, therefore, be introduced in the following sections. Please note that the term *textual model* refers to the text written using a DSL's Concrete Syntax and not to the underlying model.

2.1 Introduction to DSLs

Although DSLs usually have blurred boundaries we would like to start this chapter by presenting a definition by Martin Fowler [11].

"Domain-specific language (noun): a computer programming language of limited expressiveness focused on a particular domain. "

A DSL is a programming language and, therefore, used by humans to instruct computers to do something. It should be designed in a way that users can understand it easily, should, however, also be executable by a computer. The language nature determines that a DSL should have a sense of fluency. The expressiveness comes from expressions and the way they are composed together. A DSL has a limited expressiveness and should, therefore, restrict the amount of features to the ones needed to support its domain. DSLs are used for a particular aspect of a system rather than building an entire software system. Finally, a DSL has a clear focus and is optimized and used for tasks, which are relevant in a small domain. DSLs are usually not Turing-complete and, therefore, avoid imperative control structures such as conditions, loops, and subroutines.

A general purpose language (GPL), in contrast, is Turing-complete, which means that it can be used to implement anything that is computable by a Turing machine. Table 2.1 highlights the main differences between GPLs and DSLs.

There are two main categories of DSLs, namely *internal* and *external* DSLs [11,30].

	GPLs	**DSLs**
Domain	large and complex	smaller and well-defined
Language size	large	small
Turing completeness	always	often not
User-defined abstractions	sophisticated	limited
Execution	via intermediate GPL	native
Lifespan	years to decades	months to years (driven by context)
Designed by	guru or committee	a few engineers and domain experts
User community	large, anonymous and widespread	small, accessible and local
Evolution	slow, often standardized	fast-paced
Deprecation/incompatible changes	almost impossible	feasible

Table 2.1: General purpose languages vs domain-specific languages [30]

- **Internal DSLs** are embedded into GPLs. This means that a textual model in an internal DSL is valid code in its GPL but only uses a subset of the language's features in a particular style to handle one small aspect of the overall system. The result should have the feel of a custom language, rather than its host language. Internal DSLs are often also referred to as *fluent interface* and describe more language-like APIs. Language designers are strongly constrained by this type of DSL as any expression has to be a legal expression in the host language. Additionally, IDE support is usually missing as the IDE is not aware of the grammar, constraints, or other properties of the embedded DSL.

- **External DSLs** are languages which usually have a custom syntax and are separated from the main language of the application they work with. A textual model in an external DSL is usually processed by a parser, which interprets the language or translates it into another one. Examples of external DSLs are introduced in Section 2.3.

Due to the mentioned restrictions of internal DSLs we will only address external DSLs in this thesis.

2.2 Benefits and Challenges

In the following we summarize some of the benefits identified by Fowler [11] and Voelter et al. [30].

- **Improve development productivity.** DSLs aim at providing a means to more clearly show the intent of a part of a system. The clarity of models written in DSLs makes them easier to read. If it is easier to read a model, it is usually also easier to find and correct mistakes and modify the system. The limited expressiveness of the DSL makes it harder to say wrong things, and makes it easier to see when a user has made an error.

- **Communication with domain experts.** Communication with domain experts can be improved by providing a clear and precise language that they can understand. This does not mean that programmers should be replaced by domain experts, but that domain experts are able to read and understand models and are, ideally, able to spot mistakes. Thereby, domain experts can also see how their ideas are represented in the system. This benefit might be the most difficult to achieve but also has the broadest gain as it addresses one of the worst bottlenecks in software development — the communication between programmers and their customers.

- **Quality.** The limited expressiveness removes unnecessary degrees of freedom for programmers and can, thus, increase the quality of the created product. A well designed DSL can lead to fewer bugs, better architectural conformance, increased maintainability, and avoidance of duplicate code.

- **Validation and Verification.** Due to the higher abstraction level of DSLs, models are not cluttered with implementation details and are more semantically rich than GPLs. Analyses are easier to implement and error messages can use the domain's vocabulary. Manual reviews and validation are easier as well, as domain experts can be involved.

Of course, there are also some challenges when applying DSLs [11, 30].

- **Language cacophony.** Users often have the concern that languages are hard to learn. Using multiple languages might, therefore, be more complicated than using a single one. One should, however, keep in mind that the limited expressiveness of DSLs should make them much easier to learn then GPLs. Nevertheless, when using multiple DSLs in a project it has to be considered if learning multiple abstracting DSLs is actually easier than understanding the underlying model.

- **Effort of building the DSL.** One main challenge of using a DSL is the fact that the DSL has to be built first. Before using a DSL it should first be analyzed if the benefits of having a DSL outweigh the effort and cost of language development and maintenance. Learning curve costs of building DSLs can be amortized across multiple times that the language is used in the future.

- **Language engineering skills.** Strongly related to the previous point, building a language requires experience and skill. Although language workbenches have simplified language engineering, it has to be considered that there still is a learning curve and that building elegant languages requires experience and practice that can only be established over time.

- **DSL hell.** If language engineering and the development of new DSLs has become technically easy, there is a danger that developers create a new DSL instead of searching for and learning existing DSLs. This may lead to a collection of multiple similar but immature languages which may have overlapping domains but are still incompatible. This problem can be addressed by making DSLs incrementally extensible and effective communication in the team.

These benefits and challenges should also be considered when applying graphical representations or visualizations (discussed in Chapter 3). Each visualization can beneficially and/or disadvantageously affect a DSL.

2.3 Examples

This section presents three DSLs and should help illustrate the theoretical aspects discussed in the previous sections. DSLs are extremely versatile and frequently used in everyday software projects. All of the discussed DSLs are either directly or indirectly used in the practical part of this thesis.

CSS

Cascading Style Sheets (CSS) [1] is a DSL for adding style to web documents. An example is given in Listing 2.1.

```
h1, h2 {
  color: #926C41;
  font-family: sans-serif;
}
b {
  color: #926C41
}
*.sidebar {
  color: #928841
  font-size: 80%;
  font-family: sans-serif;
}
```

Listing 2.1: CSS example [11]

CSS is mostly used by web designers rather than by programmers. It is, therefore, a good example of a DSL which is not just read but also written by domain experts. The code example reveals CSS' declarative nature which is often used in DSLs. CSS plays a well-focused role in the web ecosystem and is used in combination with other DSLs and GPLs. The limited expressiveness of CSS is given by the absence of some features such as naming of color schemes or the lack of arithmetic functions. Another DSL which provides these missing features in the form of arithmetic operations and variables is SASS [2], which is similar to CSS and generates CSS as output [11].

CSS can be used within JavaFX for skinning components and is used for applying styles to the created visualizations of Use Case 1 and Use Case 2.

[1] w3.org/Style/CSS
[2] sass-lang.com

Graphviz

Graphviz [3] is a library for producing graphical renderings of node-and-arc graphs. Diagrams are created by defining them using the DOT language, which is an external DSL. A code example of this language is shown in Listing 2.2.

```
digraph finite_state_machine {
  rankdir = LR;
  size="8,5"
  node [shape = doublecircle]; LR_0 LR_3;
  node [shape = circle];
  LR_0 -> LR_2 [label = "SS(B)"];
  LR_0 -> LR_1 [label = "SS(S)"];
  LR_1 -> LR_3 [label = "S($end)"];
  LR_0 -> LR_2 [label = "SS(B)"];
}
```

Listing 2.2: Graphviz example [11]

Nodes can optionally be declared by using the `node` keyword. Arcs are declared using the `->` operator. Attributes can be given to nodes and arcs by using square brackets. Semicolons are optional in this language. After populating the in-memory representation of the DSL (the Semantic Model , see Section 2.4) Graphviz computes a suitable layout for the graph and renders it in various graphics formats. FXDiagram, the framework used in this thesis' practical part, uses Graphviz as a basis for the animated auto-layout feature.

Xtext's grammar language

Xtext [4] is a framework for the development of programming languages and DSLs. The DSLs created and used in the practical part of this thesis were both created using Xtext. Information about Xtext can, therefore, be found in Chapter 7. DSLs in Xtext are created by using the Xtext grammar language. It is a DSL used for the description of textual languages. The grammar language describes the language's concrete syntax and how it is mapped to an in-memory representation (the Semantic Model, see Section 2.4). The Xtext grammar language itself is implemented with Xtext. An example of a grammar defined using Xtext can be found in Listing 2.3

```
grammar org.xtext.example.mydsl.MyDsl with
                                      org.eclipse.xtext.common.Terminals

generate myDsl "http://www.xtext.org/example/mydsl/MyDsl"

Model:
  greetings+=Greeting*;

Greeting:
  'Hello' name=ID '!';
```

Listing 2.3: Xtext grammar example

[3] graphviz.org
[4] eclipse.org/Xtext

2.4 Language concepts

Before discussing the various representations of DSLs in the next chapter we would like to address the technical corner stones of DSLs in this section. There are various keywords such as abstract syntax, grammar, abstract syntax tree, semantic model, and concrete syntax which need to be clarified before talking about visualizations. The terms will be exemplified by Fowler's security system DSL [11]. The DSL is a state machine like language and uses events and states. We refer the interested reader to Martin Fowler's book for further information on the DSL. The examples should, however, be easy to follow without in-depth knowledge of the language.

Concrete Syntax (CS)

The Concrete Syntax (CS) is the notation used to illustrate the language concepts intuitively. The user interacts with the CS in order to create models. As we will show in the next chapter, the CS can be textual, graphical, symbolic, or tabular. In the remaining section and the following example we assume the CS to be textual. A textual model demonstrating the CS of the DSL is depicted in Listing 2.4.

```
events
  doorClosed D1CL
end

state idle
  doorClosed => active
end
state active end
```

Listing 2.4: Concrete syntax [11]

Grammar

A grammar formally defines the concrete syntax of a (textual) language. Note that there does not exist *the* grammar for a language but that it is possible that more than one grammar recognizes the same language. A grammar can be used to derive the *metamodel* of a language, which determines how language concepts can be combined. The grammar of the example DSL is shown in Listing 2.5, the respective metamodel in Figure 2.1.

```
root            : eventBlock stateDec*
eventBlock      : Event-keyword eventDec* End-keyword
eventDec        : Identifier Identifier
stateDec        : State-Keyword Identifier transitionDec* End-keyword
transitionDec   : Identifier Transition-Keyword Identifier
```

Listing 2.5: Grammar [11]

There are two ways of developing a DSL and linking the CS with the metamodel.

- **Grammar first.** Using this approach, first the grammar of the language is designed. The metamodel is then derived from the grammar either automatically or by providing hints in the grammar specification. This is the default approach of Xtext, where the Ecore metamodel is derived from an Xtext grammar.

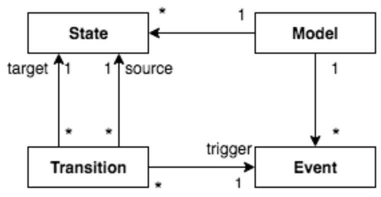


Figure 2.1: Metamodel

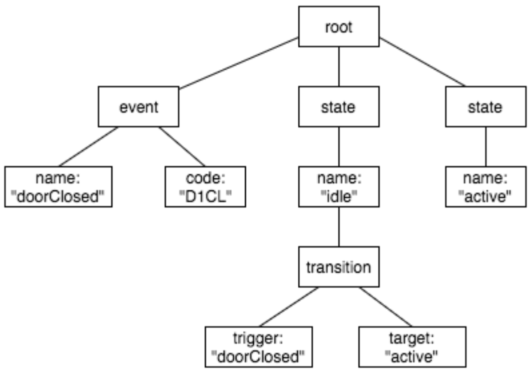


Figure 2.2: Syntax Tree

- **Metamodel first.** In the second approach, first the metamodel of the language is designed. Afterwards, the grammar is created and includes references to the already existing metamodel. It is also possible to use this approach in Xtext.

Syntax Tree

A Syntax Tree is a hierarchical representation of the textual model. It is a better structural representation for later manipulation than the words of the textual model. We can distinguish between Concrete Syntax Trees (CST) and Abstract Syntax Trees (AST). The CST retains all information of the textual model (e.g. white spaces). An AST may simplify and reorganize the input data (e.g. remove superfluous parenthesis, white space, and comments). It, however, still takes fundamentally the same form as the CST. A Syntax Tree which was generated from the textual model shown in Listing 2.4 is given in Figure 2.2.

Semantic Model

Fowler [11] defines the Semantic Model as “a representation, such as an in-memory object model, of the same subject that the DSL describes”. A DSL for describing a state machine could, for example, have a Semantic Model which is an object model and consists of objects of type state, event, etc. The textual model written using the DSL constitutes the input which ultimately populates the Semantic Model. A textual model which defines states and events would thus populate the Semantic Model with a state instance for each state and an event instance for

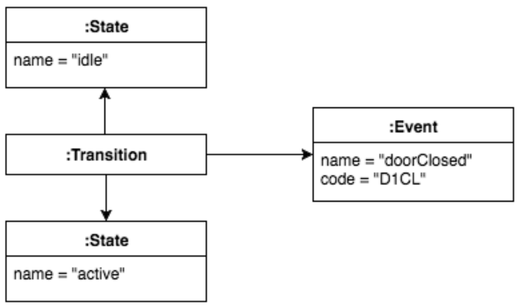


Figure 2.3: Semantic Model

each event. The language's metamodel defines the structure of the Semantic Model. A grammar includes lots of aspects that describe the input language and also implies the structure of the Syntax Tree. The metamodel, however, will be independent of any DSL used to populate the Semantic Model. The Semantic Model is, therefore, independent from the notation of concepts (i.e. the CS) and is depicted using the *Abstract Syntax* (AS). The Semantic Model of the textual model example is shown in Figure 2.3. The Semantic Model is based on what will be done with the information from a textual model and will often have a substantially different structure, which usually is not a tree structure. There are occasions when an AST is an effective Semantic Model for a DSL but these are the exception rather than the rule.

There are two ways of populating the Semantic Model [30].

- In **parser-based systems** the user only interacts with the CS and the Semantic Model is constructed from the text via a parser. In the parsing step the input (the textual model) is first transformed into a Syntax Tree. Afterwards, the parser translates the Syntax Tree into the Semantic Model. Xtext uses this parser-based approach.
- In **projectional systems** the users only see the CS but all editing gestures directly influence the Semantic Model. The CS is rendered from the Semantic Model via projection rules.

To conclude, this chapter should have given the reader an overview of DSLs, their benefits and challenges, as well as the technical corner stones of language design. The Concrete Syntax and Semantic Model are concepts which are heavily utilized when creating graphical representations of DSLs such as graphical concrete syntaxes or visualizations, which are discussed in the next chapter.

CHAPTER 3

Visualization of DSLs

This chapter introduces multiple possibilities of how users can interact with and perceive DSLs. In most cases users interact with a DSL by using a Concrete Syntax (CS). While a CS is very often of a textual nature, also graphical or tabular representations, or a mix of the two is possible. Even more possibilities arise if non-editable views such as visualizations or reports can be used. The following sections describe all of these representations in detail and show the potential of using visualizations for fostering the comprehension of DSLs.

3.1 Concrete Syntax

The design of a DSL's CS has an important impact on the user's acceptance of the language. The notation used should directly reflect the domain in order to ensure the success of the DSL.

Concerns

Voelter et al. [30] identify four main design concerns for a CS. The concerns do not only depend on the CS but also on the expressiveness of the language and the metamodel.

- **Writability.** A syntax is writable if it can be written efficiently. Conciseness, i.e. how much the user has to write, but also editing support of IDEs (e.g. code completion, quick fixes, etc.) affect the writability of a language.
- **Readability.** A syntax is readable if it can be read efficiently. It has to be considered that a very concise syntax is not necessarily also readable. This is especially the case if the writer and reader of a model are not the same person.
- **Learnability.** Learnability describes how easy a language can be learned by novice users. A syntax which uses concepts that are directly connected to the domain or IDE support (e.g. suggestions) can improve learnability.

- **Effectiveness.** A syntax is effective if it allows users to effectively express typical domain problems after they have learned the language.

Already when comparing writability and readability it becomes apparent that the design of a CS always leads to some tradeoffs. As has been mentioned, a very writable language might not be very readable or easy to learn. A very learnable language might be verbose and, therefore, lead to a syntax that is not easily writable or effective.

One way of addressing this challenge is by providing multiple CS for one language. Thereby, users can choose individually which syntax they prefer. Novice users may use a syntax which fosters comprehension and improves the learning process. In contrast, experienced users can use a syntax which focuses on writability and effectiveness.

As we will show later, besides the CS, also visualizations and reports can affect the above mentioned concerns. One of this thesis' main questions is concerned with analyzing in how far visualizations can improve the comprehension of DSLs.

Classes of CS

As previously indicated, there are multiple classes of CS [30].

- **Textual.** A textual CS uses linear textual notations and is usually based on ASCII or unicode characters. DSLs using this class of syntax look like traditional programming languages. Purely textual DSLs integrate well with existing development infrastructures, which makes their adoption easy. Text editor functionalities like copying, pasting, searching, replacing, syntax highlighting, and code completion can be used when using a textual CS. Furthermore, they are well suited for detailed or algorithmic descriptions. Referring to the concerns mentioned above, a textual language can be very effective.

- **Graphical.** Graphical DSLs use graphical shapes. Box-and-line diagrams that look like UML class diagrams or state diagrams represent an important subgroup. Graphical notations are useful when describing relationships, flow, or timing and causal relationships. They are often considered easy to learn. However, DSLs using a graphical CS may be perceived as less effective by experienced users. Building the necessary editor for a graphical CS can also involve considerable work. It has to be considered that a model is not simply 'drawn' but rather constructed in an interactive editor [31]. This editor requires features such as zooming, panning, context menus, buttons, and palettes including drag and drop functionality. An example of a graphical CS and a graphical editor based on Sirius [1] is shown in Figure 3.1.

- **Symbolic.** DSLs using a symbolic notation are textual languages having an extended set of symbols. Typical symbols are fraction bars, mathematical symbols, or subscript and superscript. Symbolic DSLs are well suited for scientific and mathematical domains that make heavy use of symbols and special notations.

[1] `eclipse.org/sirius`

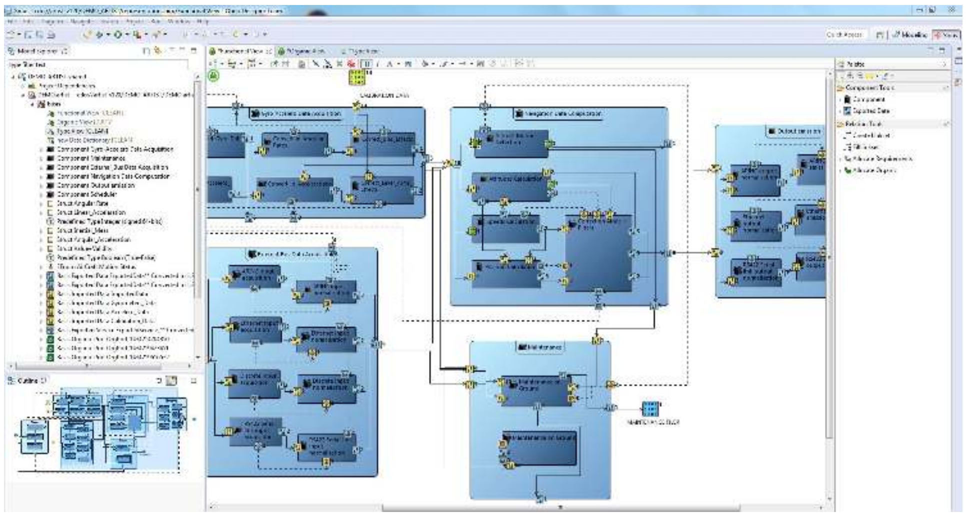


Figure 3.1: Graphical CS and editor created using Sirius [2]

- **Tabular.** Tables are useful when the CS should be able to express collections of similarly structured data items. A tabular CS can also be used for showing how two independent dimensions of data relate.

Ideally a DSL provides multiple CS and lets the user choose which one he or she prefers. Defining several notations for the same language concepts is usually easier for DSLs using projectional editing. However, in general, providing multiple syntaxes can be very work-intensive. This is especially true when developing a graphical CS: While the graphical CS might improve learnability for a DSL, defining graphical notations and especially graphical editors can be very time consuming.

Furthermore, the fact that a graphical CS mainly focuses on editing capabilities, of course, imposes restrictions on the graphical CS itself and its design. On the one hand this might impact the DSL's expressiveness as graphical editing has to be possible for all language concepts. On the other hand, as the graphical notations are mainly designed for enabling editing of the model, also the graphical language's power to improve comprehension and learnability might be restricted. We, therefore, advocate the approach of using a textual CS for editing purposes and using read-only visualizations, which will be introduced in the next section, for improving comprehension.

3.2 Visualizations

As this thesis is mainly about visualizations it is sensible to start this section with a definition. Voelter [30] defines visualizations as follows:

"A **visualization** is a graphical representation of a model that cannot be edited. It is created from the core model using some kind of transformation, and highlights a particular aspect of the source program."

Similar to the graphical CS of a DSL, a visualization has the goal of showing the model in a graphical representation. The need for a graphical representation, however, does not necessarily mean that the model has to be edited in a graphical form as well. A visualization, therefore, is read-only. This means that there has to be some alternative possibility for the user to edit the Semantic Model. Providing any class of CS, discussed in the previous section, is suitable. It is also possible to not have a CS at all and directly interact with the Semantic Model.

Young and Munro [33] identified the following desirable properties of a visualization:

- **Simple navigation with minimum disorientation.** Visualizations should be well-structured and designed in a way that users become familiar with navigating and do not get 'lost'.
- **High information content.** "Visualisations should present as much information as possible without overwhelming the user" [33]. There is a trade-off between high information content and low visual complexity.
- **Low visual complexity** Although the visual complexity greatly depends on the complexity of the information, the visual complexity should be reduced as far as possible. This often also leads to an understandable layout, well-structured visualizations, and simple navigation.
- **Varying levels of detail.** It should be possible for users to explore the visualization using varying levels of detail. A user might want to see the entirety of of the visualized model first, and investigate areas of interest in detail later.
- **Resilience to change.** Small additions or changes to the Semantic Model should not result in major changes in the visualization. A full re-positioning of elements in the visualization may result in the user becoming disoriented.
- **Good use of visual metaphors.** Metaphors use familiar concepts of the user in the visualization. Metaphors support the user in gaining understanding of the visualization.
- **Approachable user interface.** The user interface should be intuitive and flexible in order to enable simple navigation, and should avoid unnecessary overheads. Shneiderman [26] states that a user interface should support seven tasks, namely overview, zoom, filter, details-on-demand, relate, history, and extract.
- **Integration with other information sources.** Visualizations are a different representation of some information source. In most cases the visualization cannot entirely replace that source. It is, therefore, desirable to have a correlation between the visualization and its source, e.g. the source code.
- **Good use of interaction.** Allowing the user to interact with the visualization in various ways helps maintaining interest and gaining more information.

- **Suitability for automation.** Visualizations are only of practical value if automation can be applied to some extent.

The fact that a visualization cannot be edited releases developers from many restrictions. A graphical CS usually is mainly designed having editing functionality in mind. Many operations on graphical elements, therefore, have to be translated and applied to the Semantic Model. Similarly, most of the Semantic Model's elements have to be mapped to graphical elements. This bidirectional relationship may affect the design of the graphical CS and might even have an effect on the design of the language itself (i.e. the metamodel). In contrast, a visualization is directly created from the Semantic Model and, therefore, completely decoupled from the DSL. By applying some kind of transformation to the Semantic Model, developers can create arbitrary visualizations showing some interesting aspect of the model. The type of information shown in the visualization can be chosen freely and independently from any CS. Furthermore, developers can choose an arbitrary technology for implementing visualizations.

The visualization itself can take on various forms and is often automatically laid out. The Semantic Model could, for example, be transformed to a static representation, e.g. an image file. It is also possible to have fully interactive visualizations, where users can show, hide, and focus on different parts of the visualization. Interactive visualizations may also support functionality for linking elements of the visualization with another representation of the model, e.g. a textual CS in an editor. One could, for example, double-click on a graphical element in the visualization, and the respective element is automatically selected in the textual model. The other way around, i.e. selecting a section of the textual model in the editor and revealing the respective graphical element in the visualization, is possible as well.

The main reason for using visualizations is highlighting a particular aspect of the model. As previously discussed, developers are mostly unrestricted when creating visualizations for DSLs. Therefore, literature related to the general term *software visualization* can be used to identify the aspects of models which are potential candidates for visualization and the tasks where those visualizations are of actual help for users.

"Software visualization is the art and science of generating visual representations of various aspects of software and its development process" [10]. Diehl et al. [10] identified three main aspects of software:

- **Structure.** Structural aspects consist of static parts and relations of the system that can be computed or inferred without running the model. The source code, data structures, the static call graph, and the organization of the model into modules are examples for structural aspects.

- **Behavior.** Behavioral aspects refer to the execution of the model, which can be seen as a sequence of program states. A program state can contain the current code and data of the program. The execution can be viewed on different levels of abstraction.

- **Evolution.** This aspect focuses on the task of changing a software system over time. Particular focus is on issues such as reconfiguration, adaption, extension, debugging, optimization, evaluation, and project management. Moreover, the analysis of software history, i.e. multiple versions of the model, is an evolutional aspect.

As has been mentioned previously visualizing these aspects should "evoke mental images for comprehending software better" [10]. Petre et al. [22] note that "fundamentally, software visualisation is concerned with software comprehension, because comprehension underpins all stages and tasks of software development: design, debugging, maintenance and modification all require sufficient understanding of the software". Visualizations should be designed in order to help users with one or more tasks in these stages. Different tasks may require different information and, therefore, different visualizations. After reviewing existing literature in the area of software visualization we identified the following tasks, which are well applicable in the field of DSLs [10,22].

- **Design and development.** This task is concerned with the interpretation of the problem to solve, the structure of the solution, and determining if the conceptual design meets the specification.

- **Comprehension of inherited code.** As defined, comprehension is required for all tasks. Nevertheless, a separate task is defined here because, one of developers' main tasks is to understand existing models. This involves understanding how the code works, where the complexity lies and how it can be made visible.

- **Anomaly detection.** Anomaly detection is about finding quality defects and bugs in a model efficiently. Debugging may be used during the completion of this task.

- **Maintenance.** Maintenance is concerned with changing a model after delivery. Maintenance activities can be either adaptive, perfective, corrective, or preventive [18].

Maintenance tasks usually can be divided into the same stages of classical software development and, therefore, include the three tasks design and development, comprehension of inherited code, and anomaly detection. Visualizations for maintenance will basically be concerned with these three tasks and thus not separately addressed in this thesis.

Visualizations can, thus, be classified with respect to the aspect they are based on, and their goal, e.g. the user's task they should simplify. It is, of course, possible that a visualization of one specific aspect can support the user in one or more tasks. For example, a visualization showing the execution of a model could help the user with comprehension of inherited code and anomaly detection. The inversion is possible as well, meaning that a user can be supported in one task by one or more visualizations which are based on one or more aspects. For example, the task of comprehending inherited code could be simplified by visualizations of structural and behavioral aspects.

The aspects and goals of visualizations can be much more detailed than the previous example. A visualization could, for example, visualize a specific type of relationship in the source code (i.e. a structural aspect), in order to help users during the design and development by increasing the user's overview of the model.

RQ1, which is concerned with investigating if visualizations can improve comprehension, will be answered in the practical part by creating various visualizations and analyzing their capabilities of supporting users during the tasks identified above. By looking at a visualization's desirable properties and the requirement to be able to create them with as little restrictions as possible, it becomes apparent that the technology used for the implementation is very important. This is especially relevant when visualizing behavioral aspects of a DSL, as behavior is usually illustrated by means of advanced animations. As we will show in Chapter 5 the well-known frameworks for creating graphical editors are not very well suited for implementing visualizations. We, therefore, need a technology or framework which is capable of fulfilling these high requirements. JavaFX and the framework FXDiagram (both will be introduced in detail in Chapter 6) are very promising candidates for technologies and were chosen in this thesis.

This choice co-determines the structure of this thesis' practical part and leads to the work being based on two use cases. In both use cases, visualizations which should improve user's comprehension are implemented for a DSL. The first use case is a fictional one and additionally aims at exploring and illustrating the technical capabilities of JavaFX for implementing visualizations. In the second use case it is intended to evaluate the relevance of visualizations in practice. For the second use case an existing DSL of an Austrian automotive supplier will be used.

3.3 Reports

Similar to visualization, reports are used to highlight a particular aspect of the source model, while not being editable. Reports are also created directly from the Semantic Model. The main difference is that instead of using a graphical representation, reports use a textual notation [30].

Visualizations do not have to be graphical. Often simple text outputs may be enough for helping users during tasks such as debugging. Plain text output, or textual visualization in Excel can similarly to visualizations help users in understanding the underlying model and improve the communication with domain experts.

We believe that reports can be created with less effort than visualizations, however, are probably not as effective as graphical representations. The effectiveness of textual versus graphical visualizations will be addressed in RQ2.

To conclude, there are three main ways of visualizing DSLs. A CS lets users directly edit the Semantic Model. However, the bidirectional relationship between CS and Semantic Model results in a lot of effort for creating these visualizations. Additionally, from a visualization perspective, a CS restricts visualization developers. Visualizations and reports are directly created from the Semantic Model and cannot be edited. It is, therefore, much easier to add different visualizations or reports as soon as the Semantic Model is created [11]. Each visualization or report can focus on different aspects of the core model. Visualizations and reports are a good possibility if the primary users of a DSL prefer a very writable notation (e.g. a very concise textual CS), and other stakeholders would like a more readable representation.

CHAPTER 4

Use Case 1: A Game Level Design Language

As has been previously discussed, creating visualizations for DSLs requires a suitable technology. We show advantages and disadvantages of frameworks for creating graphical editors in Chapter 5. The first use case aims at exploring the capabilities of using FXDiagram, which uses JavaFX as a rendering platform. Firstly, this section introduces the scenario and language chosen for this use case, i.e. the game level design language. Secondly, the implemented visualizations are discussed with respect to the visualized aspect and the visualization's goals. Implementation details are addressed in Chapter 7.

4.1 Game level design language

As the fictional use case should mainly show the power of JavaFX we had to come up with a language providing means to do so. We chose the domain of computer games as it facilitates the application of complex geometry, animation, and simulation. The language was created using Xtext (Details on Xtext can be found in Chapter 7).

The *Level Design Language* (LDL) is used by level designers in order to conveniently define the levels of a two dimensional game. A level consists of multiple rooms which are connected by doors. Each room has a size defined by its length and width, an entrance, and an exit. An exit of one room is connected with the entry of another one. The entrance of the first room is the starting point of the level. The exit of the last room constitutes the goal of the level. The first room, and last room can be defined by the level designer.

A room can comprise multiple walls, which cannot be passed through by characters. A level designer can define walls by specifying the coordinate, i.e. row and column, where the wall should start and the coordinate where the wall should end. There usually are some kinds of enemies and dangers included in levels. The LDL enables the developer to position trapdoors and monsters. The game is over if the player steps on a trapdoor field. A monster has a certain amount of hit points and attacks with a specific speed and damage. Additionally, a monster has an aggro radius, i.e. the distance from a player at which the monster will stop its normal behavior and engage the player in combat.

A possible model of the LDL showing all important concepts is shown in Listing 4.1. The model defines three rooms including walls, trapdoors, and monsters. Please note that the detailed specification of the monsters are partly removed due to space reasons.

```
spawn => firstroom
goal => thirdroom

room firstroom {
  columns = 20
  rows = 20

  entry entry1  @ (19,0)
  exit exit1  @ (0,0) 'stairs' => secondroom

  wall wall1 from (5, 0) to (5, 12)
  wall wall2 from (0, 8) to (1,8)
  wall wall3 from (10, 15) to (19,15)

  trapdoor @ (1,1)
  trapdoor @ (10,5)
  trapdoor @ (9,9)

  monster mage1 @ (1,3) {
    hp 500
    damage 10
    speed 1.0
    range 5
    aggroradius 2
  }

  monster mage2 @ (3,7) {
    ...
  }

  monster mage3 @ (10,12) {
    ...
  }
}

room secondroom {
  columns 10
  rows 10

  entry entry2 @ (0,0)
  exit exit2 @ (9,9)  'hallway' => thirdroom

  monster mage4 @ (8,8) {
    ...
  }
}

room thirdroom {
  columns 15
  rows 15

  entry entry3 @ (0,0)
  exit exit3 @ (9,9)

  monster mage5 @ (8,8) {
    ...
  }
}
```

Listing 4.1: Level definition example using the LDL

4.2 Visualizing LDL

This section introduces the implemented visualizations for the LDL. Firstly, some consequences of using FXDiagram concerned with integration, interaction, and navigation are discussed. After that, visualizations based on static and dynamic aspects are considered.

Integration, Interaction and Navigation

Before concentrating on the specific visualizations we would like to quickly describe some basic properties of all visualizations which are predefined when using FXDiagram.

As stated above, visualizations should have a good use of interaction and should have some sort of integration with other information sources. When using FXDiagram, the user is provided with two views which are shown side by side when opening the IDE. The first view, is the textual editor which can be used to edit the model. The second view shows the graphical visualization. Visualizations can be opened by right-clicking on an element in the editor and selecting the desired visualization. Visualizations which are only useful in the context of another visualization can be opened using the graphical context menu, which can be opened by right-clicking somewhere in the graphical view. By double-clicking elements in the graphical view, the corresponding element in the textual view is selected.

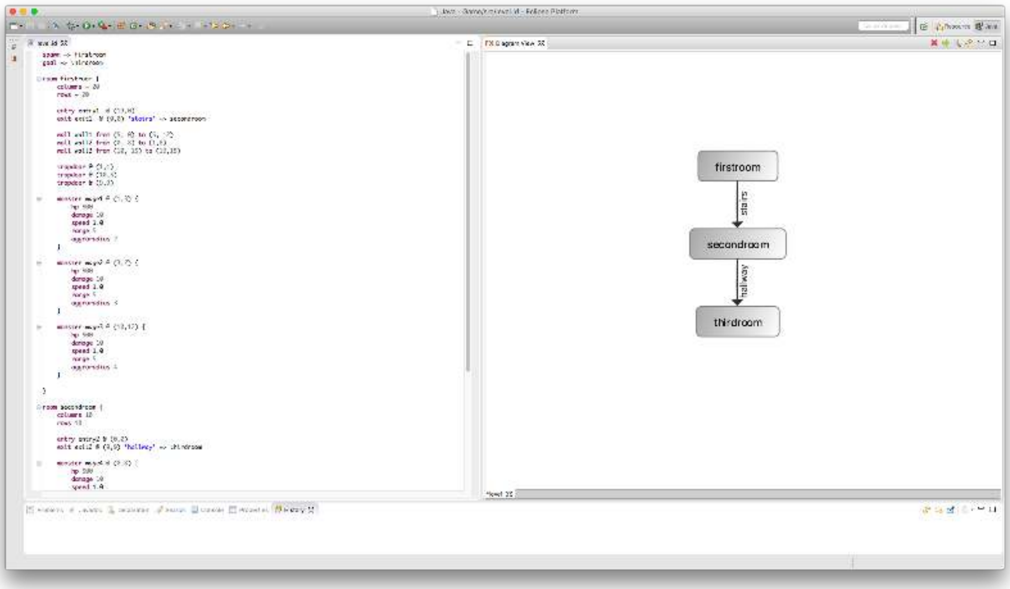
Users can interact with the graphical representation intuitively by either using the graphical context menu, keyboard shortcuts, or touch gestures. FXDiagram provides features such as zooming, auto-layout, navigation, undo and redo, and exporting graphical representations in image format out of the box.

Static aspects

As described in Section 3.2, visualizations can illustrate a model's structural aspects. One of the desired properties of visualizations is to provide the user with varying levels of detail. The first visualization thus intends to give the user an overview of the models entirety. The level is visualized by showing the rooms as nodes. Two nodes are connected if the respective rooms are connected by a door. The visualization is shown in Figure 4.1.

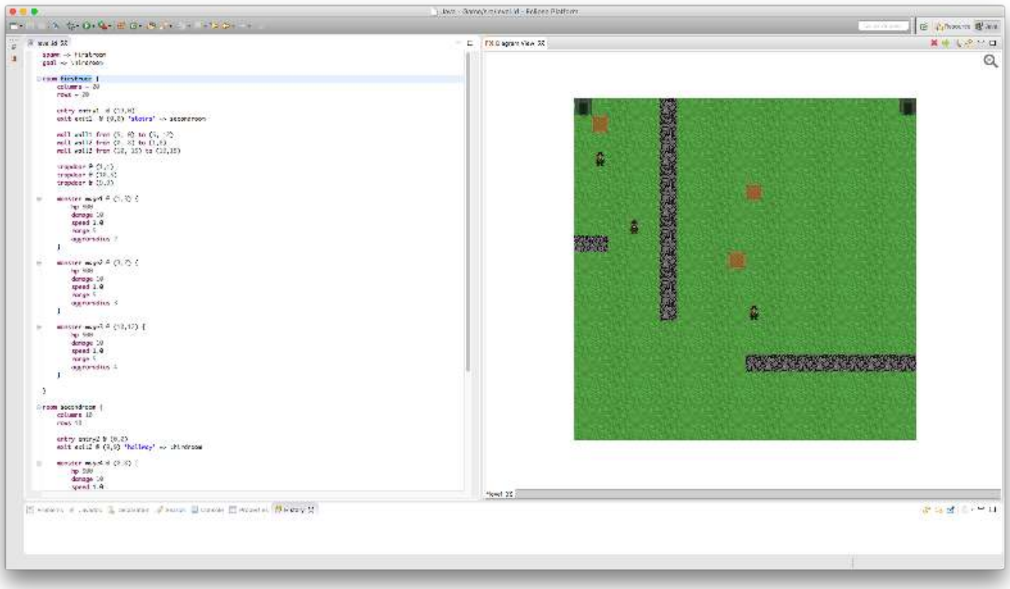
By double-clicking on one of the nodes, a more detailed representation of the respective room such as depicted in Figure 4.2 is shown. Images are used for creating the background and for representing language elements such as entries and exits, trapdoors, monsters, and walls. Double-clicking on any of the elements selects the respective element in the editor. This visualization intends to give the user a more detailed structural overview of a room and the positioning of its elements. While the textual CS lets the user quickly create new rooms, it might be easier for the user to assess the correct positioning, the distance between elements, and the correct definition of walls in the graphical view.

It is often helpful to graphically add supplementary information to an existing visualization. While the detailed room visualization gives the user a great overview, it is hard for the level designer to assess which of the monsters a player, going from the entry to the exit, would have



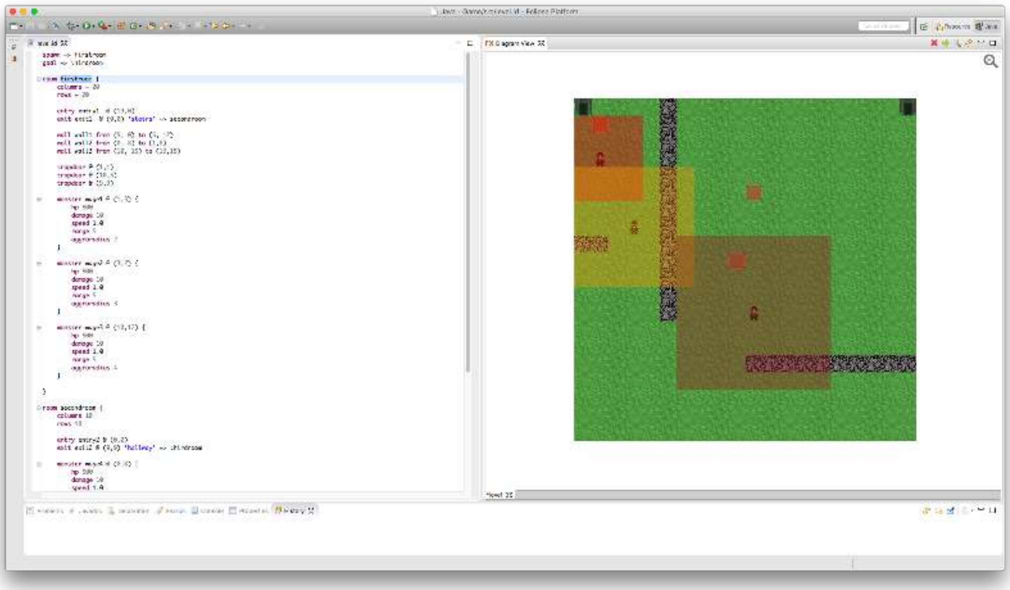
```
spawn -> firstroom
goal -> thirdroom

room firstroom {
    columns = 20
    rows = 20

    entry entry1 @ (13,0)
    exit exit1 @ (0,2) 'stairs' -> secondroom

    wall wall1 from (5, 0) to (5, 12)
    wall wall2 from (2, 8) to (1,8)
    wall wall3 from (10, 15) to (19,15)

    trapdoor @ (1,1)
    trapdoor @ (10,5)
    trapdoor @ (9,3)

    monster mage1 @ (1,3) {
        hp 500
        damage 10
        speed 1.0
        range 5
        aggroradius 7
    }

    monster mage2 @ (3,7) {
        hp 500
        damage 10
        speed 1.0
        range 5
        aggroradius 3
    }

    monster mage3 @ (10,12) {
        hp 500
        damage 10
        speed 1.0
        range 5
        aggroradius 1
    }
}

room secondroom {
    columns 10
    rows 10

    entry entry2 @ (0,2)
    exit exit2 @ (9,9) 'hallway' -> thirdroom

    monster mage4 @ (9,8) {
        hp 500
        damage 10
        speed 1.0
```

Figure 4.1: Level overview



```
spawn -> firstroom
goal -> thirdroom

room firstroom {
    columns = 20
    rows = 20

    entry entry1 @ (13,0)
    exit exit1 @ (0,2) 'stairs' -> secondroom

    wall wall1 from (5, 0) to (5, 12)
    wall wall2 from (2, 8) to (1,8)
    wall wall3 from (10, 15) to (19,15)

    trapdoor @ (1,1)
    trapdoor @ (10,5)
    trapdoor @ (9,3)

    monster mage1 @ (1,3) {
        hp 500
        damage 10
        speed 1.0
        range 5
        aggroradius 7
    }

    monster mage2 @ (3,7) {
        hp 500
        damage 10
        speed 1.0
        range 5
        aggroradius 3
    }

    monster mage3 @ (10,12) {
        hp 500
        damage 10
        speed 1.0
        range 5
        aggroradius 1
    }
}

room secondroom {
    columns 10
    rows 10

    entry entry2 @ (0,2)
    exit exit2 @ (9,9) 'hallway' -> thirdroom

    monster mage4 @ (9,8) {
        hp 500
        damage 10
        speed 1.0
```

Figure 4.2: Detailed view of a room



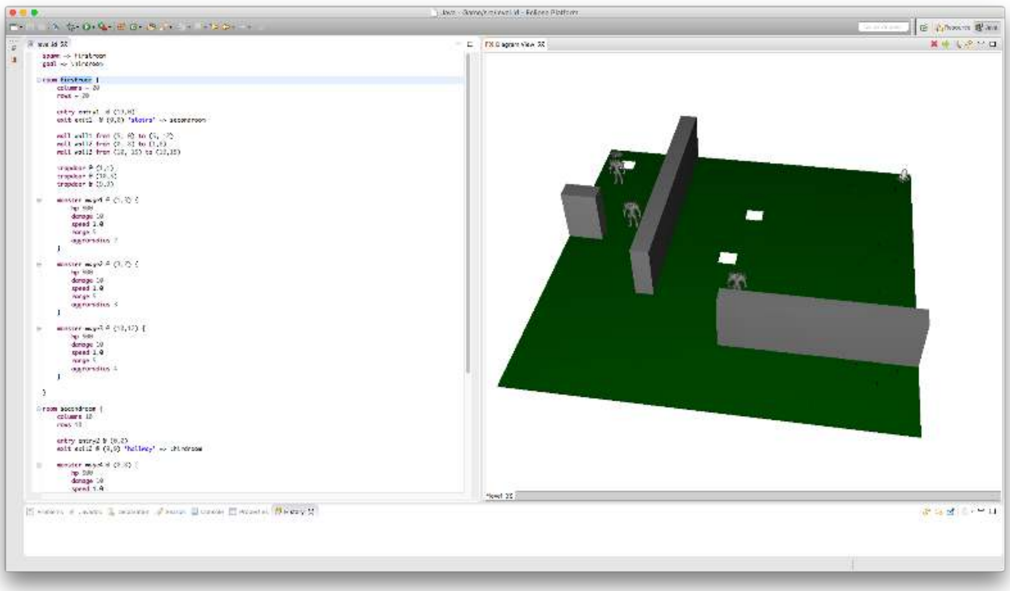
```
spawn -> firstroom
goal -> thirdroom

room firstroom {
    columns = 20
    rows = 20

    entry entry1 @ (13,0)
    exit exit1 @ (9,0) 'alcatra' -> secondroom

    wall wall1 from (5, 0) to (5, 17)
    wall wall2 from (2, 8) to (1,8)
    wall wall3 from (10, 15) to (20,15)

    trapdoor @ (1,2)
    trapdoor @ (10,5)
    trapdoor @ (1,3)

    monster mxy1 @ (1,3) {
        hp 500
        damage 10
        speed 1.0
        range 5
        aggroradius 7
    }

    monster mxy2 @ (3,7) {
        hp 500
        damage 10
        speed 1.0
        range 5
        aggroradius 5
    }

    monster mxy3 @ (10,12) {
        hp 500
        damage 10
        speed 1.0
        range 5
        aggroradius 4
    }

}

room secondroom {
    columns 10
    rows 10

    entry entry2 @ (0,2)
    exit exit2 @ (9,9) 'hallway' -> thirdroom

    monster mxy4 @ (9,8) {
        hp 500
        damage 10
        speed 1.0
```

Figure 4.3: Detailed view of a room including the monsters' aggro radius

to fight, i.e. which monsters' aggro radius the player has to enter. This is very difficult to find out by looking at the source code only, but not even obvious with the additional detailed room view. The information about the aggro radius can, however, be highlighted by a very simplistic visualization which is added on top of the detailed room view. The visualization is activated through the graphical context menu and shown in Figure 4.3. Looking at the visualization it can easily be observed that a player going from the entry to the exit has to fight at least two monsters.

In the context of the room visualization, a second very simple additional visualization was implemented which has the goal of improving the representation of the strength of monsters. Again, by activating the visualization using the graphical context menu, the size of each monster is adjusted in a way that it correlates to the monster's strength, i.e. health points, damage, and speed. This visualization, of course, can be combined with the aggro range visualization. A combination of multiple visualization can give the user a comprehensive overview of multiple aspects and may help level designers even more than individual views in some situations.

In order to explore the possibilities developers have using JavaFX and FXDiagram, we also developed an experimental visualization. While the LDL is primarily used for defining two dimensional levels, this does not mean that visualizations need to be two dimensional as well. Figure 4.4 shows our results in form of a three dimensional visualization of a room. This gives a level designer a completely new perspective and allows for inspecting the defined room in a very different way. FXDiagram by default does not support 3D content and there are a lot of challenges that have to be addressed when implementing 3D visualizations. The technical de-



```
spawn -> firstroom
goal -> [illegible]

room firstroom {
    columns = 20
    rows = 20

    entry entry1 @ (13,0)
    exit exit1 @ (9,0) 'stairs' -> secondroom

    wall wall1 from (5, [illegible]) to (5, [illegible])
    wall wall2 from ([illegible], [illegible]) to ([illegible])
    wall wall2 from (10, 15) to (18,15)

    trapdoor @ ([illegible])
    trapdoor @ (10,[illegible])
    trapdoor @ ([illegible])

    monster [illegible] @ ([illegible]) {
        hp 500
        damage 10
        speed 1.0
        range 5
        aggroradius [illegible]
    }

    monster [illegible] @ ([illegible]) {
        hp 500
        damage 10
        speed 1.0
        range 5
        aggroradius [illegible]
    }

    monster [illegible] @ (10,17) {
        hp 500
        damage 10
        speed 1.0
        range 5
        aggroradius [illegible]
    }
}

room secondroom {
    columns 10
    rows 10

    entry entry2 @ (0,2)
    exit exit2 @ (9,9) 'hallway' -> thirdroom

    monster [illegible] @ ([illegible]) {
        hp 500
        damage 10
        speed 1.0
```

Figure 4.4: 3D visualization of a room

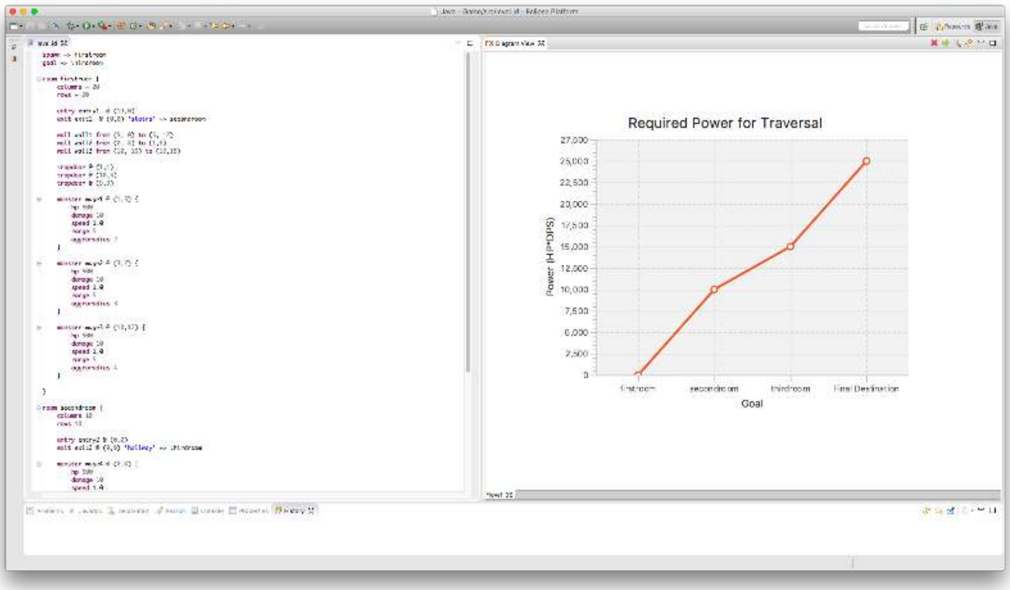
tails and found limitations of FXDiagram are described in Chapter 7.

Finally, it is also possible to analyze the structure of models and derive information that is relevant for dynamic aspects. Level designer might, for example, be interested in the difficulty of a designed level. The difficulty in the LDL is mainly defined by which and how many monsters a player has to fight in order to successfully traverse the level, i.e. go from the entry of the first room to the exit of the last one. This difficulty can easily be computed and analyzed by looking at the structure of the model. Using another kind of visualization, this data is presented in the form of a chart such as shown in Figure 4.5. The chart shows how much power is required for reaching a specific goal in the game, e.g. completing the first room, or completing the whole level.

It already becomes apparent when looking at the previous examples that information can be shown in different forms and that visualizations can be very diverse. Some visualizations are simplistic node-and-edge-diagrams, others require advanced graphics, and again others can be simple charts. A very powerful base technology, offering all those possibilities, is vital for developers for creating the best-suitable visualizations.

Dynamic aspects

Nelson [21] states that game designers "have mental models of how the game they're designing should work, and spend considerable time mentally tracing through possibilities". Analyzing and visualizing the model should speed up this process and support level designers. Nelson



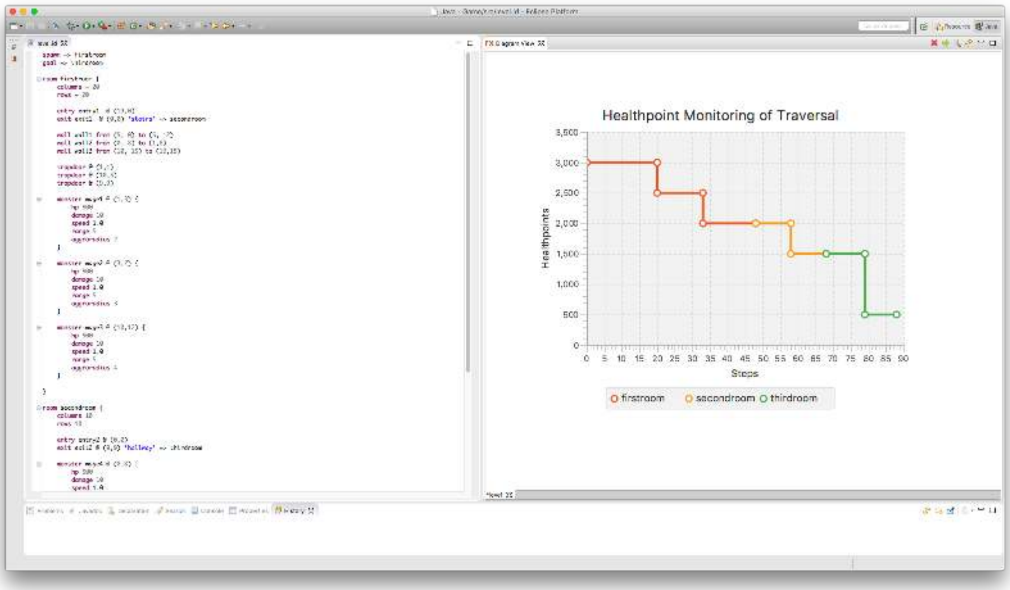
```
spawn -> firstroom
goal -> [illegible]

room firstroom {
    columns = 20
    rows = 20

    entry entry1 @ (13,0)
    exit exit1 @ (9,0) 'akaira' -> secondroom

    wall wall1 from (5, 0) to (5, [illegible])
    wall wall2 from (2, [illegible]) to (1,8)
    wall wall3 from (10, 15) to (12,15)

    trapdoor @ (1,[illegible])
    trapdoor @ (10,5)
    trapdoor @ (5,3)

    monster mey1 @ (1,3) {
        hp 500
        damage 10
        speed 1.0
        range 5
        aggroradius 7
    }

    monster mey2 @ (3,7) {
        hp 500
        damage 10
        speed 1.0
        range 5
        aggroradius 5
    }

    monster mey3 @ (13,17) {
        hp 500
        damage 10
        speed 1.0
        range 5
        aggroradius 4
    }
}

room secondroom {
    columns 10
    rows 10

    entry entry2 @ (0,2)
    exit exit2 @ (3,9) 'halley' -> thirdroom

    monster mey4 @ (9,[illegible]) {
        hp 500
        damage 10
        speed 1.0
```

Figure 4.5: Visualization showing the required power for traversing the level

identifies several strategies for analyzing games. Answering questions such as ‘Is this possible?’, ’How is this possible?’ and analyzing necessities and dependencies can greatly help level designers. The answers to these questions, and the results of analyses can, of course, be presented using visualizations.

‘Is it possible for a player to complete the level?’ is an obvious question a designer using LDL could ask him- or herself. By initially providing the player stats this question can be answered by looking at the model again. The player stats are gathered by means of a form which is shown prior to the visualization. For this visualization a slightly more complex computation is required as the optimal path for traversal has to be found and fights with monster have to be simulated. After these computations, however, a visualization with a very high information content can be created. The visualization in Figure 4.6 shows the player’s health points during a traversal. Using this visualization a designer can on the one hand see if it is possible to complete the level. On the other hand, one can also see how monsters are distributed over the rooms. A consistent monster distribution might be preferred over one with many empty rooms.

Another interesting question for level designers is ‘How is it possible to achieve X?’. If, for example, it is possible for a character to traverse a level, it might be interesting how it can exactly traverse it. Ideally, a designer would like to see the exact path, the monsters the player has to fight, and how the health points change. This is, therefore, already very similar to a full game simulation. As soon as the path is calculated this simulation can, however, easily be implemented utilizing the animation features of JavaFX. Animation features are vital for visualizations of be-



```
spawn -> firstroom
goal -> thirdroom

room firstroom {
    columns = 20
    rows = 20

    entry entry1 @ (13,0)
    exit exit1 @ (0,2) 'stairs' -> secondroom

    wall wall1 from (5, 0) to (5, 17)
    wall wall2 from (7, 8) to (7,0)
    wall wall3 from (12, 15) to (12,15)

    trapdoor @ (1,7)
    trapdoor @ (10,5)
    trapdoor @ (5,3)

    monster mage1 @ (1,3) {
        hp 500
        damage 10
        speed 1.0
        range 5
        aggroradius 7
    }

    monster mage2 @ (3,7) {
        hp 500
        damage 10
        speed 1.0
        range 5
        aggroradius 3
    }

    monster mage3 @ (10,17) {
        hp 500
        damage 10
        speed 1.0
        range 5
        aggroradius 1
    }
}

room secondroom {
    columns 10
    rows 10

    entry entry2 @ (0,2)
    exit exit2 @ (9,9) 'hallway' -> thirdroom

    monster mage4 @ (3,3) {
        hp 500
        damage 10
        speed 1.0
```

Figure 4.6: Health points during traversal

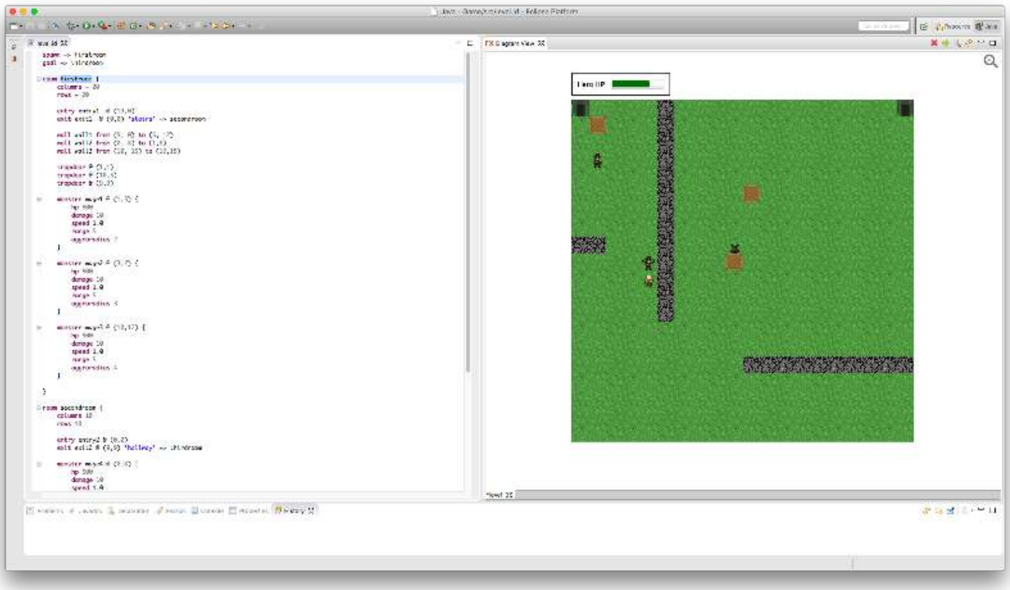
havior. This is one of the main reasons why existing frameworks for creating graphical editors are not suitable for creating visualizations. While we address implementation details again in Chapter 7, the traversal requires the parallel execution of two or more animations. The movement of the player, for example, is created by simultaneously playing a sprite animation (the walk cycle), and moving the player in the room. Animations are also required for fighting and dying sequences. The traversal is shown on top of the detailed room view and is started after specifying the player's stats and the simulation speed. Additionally, a progress bar showing the player's health points is inserted. Figure 4.7 shows the described traversal visualization.

The traversal has also been implemented in the three dimensional view, which from a technical point of view could be done very similarly to the two dimensional traversal.

Summary

To sum up, Use Case 1 shows that depending on the type of information to be visualized, the types of visualization can greatly vary. By simply trying to find helpful visualizations for level designers we came across node-and-edge-diagrams, views mainly relying on graphics, charts, three dimensional views, forms, and visualizations heavily utilizing animations. Our assumption that visualization can take on very different forms was, therefore, correct.

Use Case 1 also has the goal of showing the capabilities of using FXDiagram and JavaFX for implementing these very diverse visualizations. We can confirm that FXDiagram in fact only adds a very thin abstraction layer on top of JavaFX. FXDiagram makes it easy to define basic node-and-edge diagrams and the framework greatly helps developers when defining integration



```
spawn -> firstroom
goal -> [illegible]

room firstroom {
    columns = 20
    rows = 20

    entry entry1 @ (13,0)
    exit exit1 @ (9,0) 'stairs' -> secondroom

    wall wall1 from (5, 0) to (5, 12)
    wall wall2 from (0, 8) to (1,8)
    wall wall3 from (10, 15) to (19,15)

    trapdoor @ (1,1)
    trapdoor @ (10,5)
    trapdoor @ (5,3)

    monster mage1 @ (1,3) {
        hp 500
        damage 10
        speed 1.0
        range 5
        aggroradius 7
    }

    monster mage2 @ (3,7) {
        hp 500
        damage 10
        speed 1.0
        range 5
        aggroradius 5
    }

    monster mage3 @ (12,17) {
        hp 500
        damage 10
        speed 1.0
        range 5
        aggroradius 1
    }

}

room secondroom {
    columns 10
    rows 10

    entry entry2 @ (0,2)
    exit exit2 @ (9,9) 'hallway' -> thirdroom

    monster mage4 @ (2,3) {
        hp 500
        damage 10
        speed 1.0
```

Figure 4.7: Room traversal simulation

behavior between the textual and graphical views. At the same time, FXDiagram, does not get into the way when language specific graphical visualizations have to be created. JavaFX makes it very easy to use forms, layouts, images, 3D content, charts, and animations. The first use case, thus, already indicates that JavaFX and FXDiagram can be well-suited technologies for implementing visualizations. We refer the interested reader to Chapter 6 for the detailed visualization architecture and Chapter 7 for implementation details.

CHAPTER 5

Existing Visualization Frameworks

This chapter discusses the most well-known frameworks for creating diagram editors and views. The first sections introduce GEF3, GMF, Graphiti, and Sirius. A discussion including advantages and disadvantages of these technologies in the context of visualizations is provided in Section 5.5.

5.1 GEF3

The Graphical Editing Framework (GEF) [1] [20, 24] was contributed by IBM in 2002. GEF was initially composed of two components, namely Draw2d and GEF (MVC). Draw2d is a 2D rendering framework and is a lightweight extension to SWT. It does not provide any interactive behavior and is solely used for displaying graphical information on an SWT canvas. Draw2d may be used stand-alone or as a visualization technology for GEF (MVC). GEF (MVC) is a framework used for implementing SWT-based tree editors and Draw2d-based graphical editors and views. It provides features such as a palette, drag-and-drop support, a command stack for undoing and redoing, and support for printing. GEF (MVC) uses the Model-View-Controller pattern.

GEF3 was introduced in 2004 and offered most of the features that make up GEF including rulers and guides, grid, snap-to-geometry, cloning, panning, fly-out palette, and shortest path connection routing. In 2007 Zest was added to GEF as a third component. Zest is a visualization toolkit based on SWT and Draw2d and provides a JFace-like interface for binding a Java model with a Draw2d diagram. In 2009 support for SWT line style attributes and advanced graphics support were added to Draw2d. Zest added curved connections and offered nested contents. Releases in 2010 and 2011 added support for improved clipping and scrollable feedback as well as refactorings. A release in 2012 included mostly bug-fixes and clean-ups, but no significant innovations. The development of Zest 2 was started in 2010 and GEF4 was initiated in 2011. We refer the reader to Chapter 6 for further information on GEF4.

[1] `eclipse.org/gef`

EditParts are the central elements in GEF applications and play the role of the controllers that specify how model elements are mapped to visual figures and how these figures behave in different situations. A figure is a graphical view, implemented in the lower-level Draw2d framework. Figures can have arbitrary, non-rectangular shapes, can be nested, can be transparent or opaque, and can be ordered into layers. Usually an EditPart is created for each model element, which leads to a hierarchy similar to that of the model. There are two main types of EditParts related to diagrams, namely GraphicalEditParts and ConnectionEditParts. The first provide a graphical representation for their model element, while the latter represent connections between GraphicalEditParts. Most EditParts require an EditPolicy, which turns event request into commands. GEF uses a command pattern to implement an undo stack, i.e. each command has to store its own undo information and implement an undo method. This leads to the following communication chain: A user executes an operation, e.g. delete, by interacting with a tool. The tool creates a request and forwards it to an EditPart. The EditPart does not process the request itself but forwards it to an EditPolicy. The EditPolicy then creates a command which will be executed in order to fulfill the request. A command can finally modify the model. EditParts have to monitor the model for changes. When a change is observed, the EditParts have to refresh their visual representation [20].

GEF3 gives developers a lot of freedom when developing graphical editors and views. This freedom, however, comes at a high price. Developers have to understand the essentials of GEF's complex architecture and a lot of technical details about the underlying communication chain. Furthermore, the rendering platform Draw2d does not provide any behavior or higher-level geometric abstractions which may result in a lot of effort for implementing very basic things. Moreover, Draw2d is integer-based, which may result in rendering errors, and does not support complex geometry (e.g. Bézier curves). The frameworks discussed in the following build on GEF3 and aim at offering developers higher-level abstractions in order to reduce the effort for creating graphical editors and views.

5.2 GMF

GEF laid the basis for the development of graphical editors. However, understanding the complex framework was very time-consuming and led to uncomfortable development as well as redundant code. GEF enabled the use of custom models, which resulted in developers often using Plain Old Java Objects (POJOs). POJOs, however, lack functionality such as serialization and the ability to listen to model changes. The Eclipse Modeling Framework (EMF) provided serialization and listening to model changes out of the box. It was, therefore, a logical step to integrate EMF models with GEF. The Graphical Modeling Framework (GMF) [2] was first released in June 2006 and aims at providing a generative bridge between EMF and GEF.

The GMF project can be split into the tooling and runtime components. GMF Tooling provides a model-driven approach to generate graphical editors in Eclipse. By defining a tooling-, graphical-, and mapping definition, one can generate a fully functional graphical editor based on the GMF Runtime. GMF Runtime offers a set of reusable components for graphical editors, such as printing, image export, actions, and toolbars.

[2] `eclipse.org/modeling/gmp`

A set of models needs to be created in order to generate a graphical editor [3]. GMF supports wizards for creating these models and provides a tree-editor for defining the specific models.

- **Graphical definition.** The graphical definition defines the visual aspects of the generated editor. Figures can be created using abstractions for rectangles, rectangles with rounded corners, ellipses, or polygons, and can be further customized using multiple properties (e.g. fill, line kind, line width, etc.). Using layout classes, figures can be combined in order to create more complex figures without directly accessing underlying GEF components. Besides figures, also diagram nodes and connections can be created. A diagram node refers to a figure. Figures can be reused because multiple nodes can refer to the same figure.

- **Tooling definition.** This definition comprises things related to the editor palettes, menus, etc. A palette can easily be created again in a tree-editor. Tools, e.g. a creation tool, can be added to the palette. A creation tool can be further specified by a name and an image.

- **Mapping definition.** The mapping definition is used to define the mapping between the EMF model and the visual model (graphical definition and tooling definition). Again, a wizard can be used for defining these mappings. After defining the diagram's root element, the individual model elements can be mapped to visual elements.

After creating these three definitions the graphical editor can be generated. This results in a new Eclipse plugin project which can be launched in a new Eclipse runtime workbench. There are, of course, much more features included in GMF. Diagrams can be validated, for example by allowing connections only between specific types of nodes. Collapsed and expanded compartments are supported as well as animated zoom and layout, image export, diagram assistants, and direct editing (e.g. text inside the diagram).

GMF makes it much easier for developers to create graphical editors by providing higher-level abstractions and hiding most of the more complex aspects of GEF. The framework can, of course, not fully compensate the previously discussed problems of Draw2d. This means that rendering problems may also occur when using GMF.

5.3 Graphiti

Graphiti [4] was released in 2010 by SAP, targeting similar objectives as GMF, i.e. enabling the easy development of graphical diagram editors. Graphiti provides a plain Java API for building graphical tools, tries to limit the dependencies to an absolute minimum, and provides the ability to use any existing layout algorithms for autolayouting a diagram. Several differences between GMF and Graphiti were defined in the proposal of Graphiti [5]. These differences are shown in Table 5.1.

[3] `ibm.com/developerworks/library/os-ecl-gmf`
[4] `eclipse.org/graphiti`
[5] `eclipse.org/proposals/graphiti`

	Graphiti	**GMF**
Architectural concept	runtime centric API	generative
API	self-contained	refers to GEF functionality
Client logic	centralized (feature concept)	functionality distributed since there are no constraints
Look & Feel	sophisticated defaults defined by usability specialists (highly customizable according to the tool requirements)	simple defaults (highly customizable according to the tool requirements in generated coding)

Table 5.1: Differences between Graphiti and GMF [1]

Just like GMF, Graphiti is based on GEF and Draw2d and also supports EMF on the domain side. Diagrams are described using a metamodel and diagram data is strictly separated from the domain data. This allows to render diagrams in different environments using various engines. Graphiti tries to completely hide GEF and Draw2d, which means that users write plain Java code when building an editor. Knowledge about the base technologies is not required.

From a technical point of view, developers write so-called *features* to add functionality to the graphical editor. The complete life cycle (creating, editing, renaming, moving, deleting, etc.) of model elements and their graphical representations is implemented by means of features. Default-features are provided by the framework and can later be replaced or extended with special behavior. Graphiti consists of two main components namely, the interaction component and the diagram type agent. The user interacts with the interaction component (provided by Graphiti), to manipulate models. The diagram type agent consists of the code written by the developer. It provides tool behavior providers which define how tools behave in specific situations. Tools can influence, for example, what will be displayed in the editor, how selections, double-clicks, and zooms are handled, and which context menus and context buttons are available. Context buttons appear around the currently hovered shape and can be used to trigger object specific operations. Operations are defined by the tools by implementing features [29].

Overall, Graphiti is very similar to GMF. It is based on GEF and Draw2d, and tries to provide abstractions to simplify the development of graphical editors. Rendering issues caused by Draw2d which were previously discussed may occur here as well. The focus on usability and better visual defaults is an important step towards improving the user experience, which is very important for visualizations in general.

5.4 Sirius

Sirius [6] is a framework for creating graphical modeling workbenches that was first released in 2014. Sirius is based on EMF and GMF and is, therefore, on the highest abstraction level of the frameworks discussed so far. Sirius aims at providing specific multi-view workbenches through graphical, table, or tree modeling editors. All of those modeling editors are synchronized which

[6] eclipse.org/sirius

means that models can be edited and viewed in all viewpoints. As opposed to other frameworks, users should be able to define their own modeling workbenches even with very little technical knowledge and knowledge of Eclipse. However, at the same time workbenches should be deeply customizable.

A Sirius modeling tool is defined by means of a configuration file in a Viewpoint Specification project. A tree-editor and property views can be used to edit the configuration file. The configuration can be used to style model elements, add behavior and navigation tools, add additional layers, and add viewpoints such as tables, matrices, and trees. The definition of the modeling tool is interpreted at runtime, which provides the developer with instant feedback.

Sirius offers many high level features which should help developers in quickly creating new workbenches. Nodes can be rendered using predefined graphical shapes, images, and custom shapes. Nodes can be connected by two types of edges. Element-based edges can be used when a semantic model element exists which represents the connection. Relation-based edges define the source and target mappings, i.e. at which types of graphical elements the edges start and end, and a Target Finder Expression. The Target Finder Expression is evaluated in the context of the source elements and should return the target elements. Elements can be structured using containers and compartments. Sirius also provides bi-directional links between the model-browser and diagram view. The release of Sirius 4.0 in June 2016 introduced new features such as support for internationalization, better SVG rendering, and user-configurable filters for hiding diagram elements according to rules.

All in all, Sirius aims at providing many advanced features for creating modeling workbenches with a minimum amount of effort. While this may significantly speed up the development process, it may also restrict developers due to the high level abstractions. Again, rendering issues may occur because of the underlying base technologies.

5.5 Discussion of Existing Approaches

Koehnlein [15] identified a mismatch between diagram editor frameworks and the final product users want.

Diagram editor frameworks are designed for developers and provide high level abstractions. Diagram editor frameworks aim at giving developers a maximum amount of features with minimum effort. Furthermore, diagram editors based on GEF3, provide standard behavior and often expose rendering issues.

However, the product the user wants, should be specifically targeted at users of the language. The user wants a custom — not generic — solution. The product should focus on usability, should provide great visuals, and should also be fun to use.

Therefore, developers have to solve a very specific (not generic) use case which requires a custom solution. Generic high level abstractions can only be provided for a unified use case. This use case, however, usually does not exist and visualizations always require a lot of customization. This means that from a user's perspective developers should not use a diagram editor framework because their use case is very specific. In a specific use case the higher level abstractions of diagram editor frameworks can greatly restrict developers. Often developers have to go beyond the levels the abstractions try to hide in order to customize some details. Sometimes changes might not even be possible at all. The restrictions imposed by diagram editor frameworks also have a direct effect on the usability. Behavior and metaphors are often predefined in diagram editors. This leads to the misunderstanding that usability is automatically addressed by the framework. Usability, however, is something that cannot be provided out of the box and usually has to be addressed by the developer. Customization and usability are also very important when developing visualization.

To sum up the previous sections, Sirius, Graphiti, and GMF are diagram editor frameworks used for building graphical editors. All of these frameworks are based on GEF3. On the one hand, GEF3 is on the lowest possible abstraction level and, therefore, does not impose limitations on developers and enables customization. However, it is hard to use and limited in respect to rendering capabilities. On the other hand, frameworks on a higher abstraction level such as Sirius, Graphiti, and GMF are easier to use, but try to solve a generic problem and thus restrict developers. The restrictions of higher-level frameworks and the limitations of the underlying technology GEF3 discussed in Section 5.1 do not allow for the development of modern visualizations. The discussed diagram editor frameworks, thus, do not seem to be reasonable technologies for implementing visualizations. Developing visualizations completely without a framework, however, is a very complex and costly task. This does not appear to be an option either.

Koehnlein addresses this problem by developing FXDiagram, a framework which uses JavaFX as a rendering platform. It adds only a thin layer on top of JavaFX which provides abstractions for nodes and edges. Developers can arbitrarily customize visualizations utilizing the capabilities of JavaFX. At the time of working on this thesis, FXDiagram appeared to be the best framework for developing visualizations and is, therefore, used in this thesis' practical part. FXDiagram is described in detail in Chapter 6.

CHAPTER 6

Visualizations based on JavaFX

This chapter introduces basic concepts that are important when implementing visualizations. Firstly, it is explained why unidirectional views are usually easier to implement than bidirectional ones. We highlight why usability and aesthetics are of major importance when implementing modern visualizations and that a proper rendering technology is necessary. After that we introduce JavaFX as a base technology, and frameworks that are based on JavaFX. Finally, we present a technology-independent visualization architecture which could be defined after the experience gained from implementing the visualizations of Use Case 1.

6.1 Diagram Editors vs Diagram Views

While Chapter 5 analyzed the properties of diagram editor frameworks, this section discusses some issues concerned with the overall idea of a diagram editor.

Providing both graphical and textual editing is a very complex task. Koehnlein highlights in a talk given in 2015 [16] several technical challenges that emerge when trying to provide both in parallel. Koehnlein also states that solutions to these challenges always include some kind of trade-off, and that these trade-offs often are trade-offs in the user experience. The challenges are described in the following.

- **Cross-reference semantics.** When referring to an element in text one usually uses its name. When referring to elements in a diagram a technical ID may be used. After changing a name in a diagram, one expects that some property of an object is changed. When changing a name in text, users are required to execute a rename refactoring operation in order to update all the cross references correctly. Therefore, if rename behavior is implemented correctly, a name change in the diagram should trigger a rename operation on the textual side. This operation could, however, fail. The operation could fail because of introducing ambiguities in a completely different part of the model. A common solution is to display a dialog which informs the user that there was a problem during the refactoring operation. The user, however, is usually not even aware of the fact that he or she is performing a refactoring. This is a major usability issue that cannot easily be solved without negatively affecting the user experience.

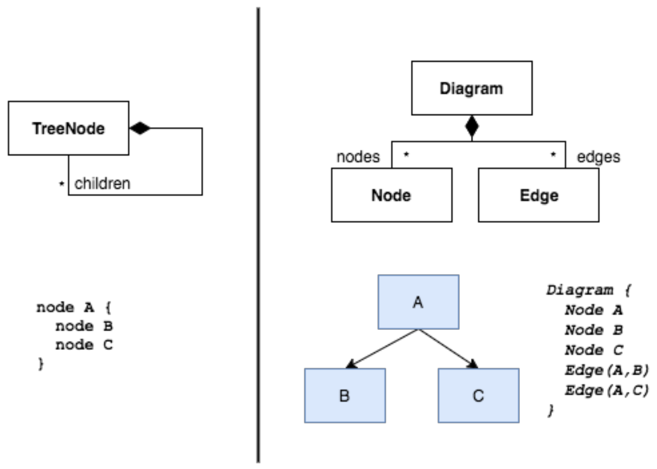
- **Identity and transaction.** Identities and the lifecycle of objects is usually very different on the textual and the graphical side. In the diagram, objects are usually long-living, while objects in the textual side are short-living and generated while parsing. Having direct references between these in order to edit both at the same time can be very difficult. This also boils down to marrying two transactions frameworks that have very different semantics.

- **Persistence.** The previous challenge also yields the question of when to synchronize models from the graphical and textual side. One possibility is to synchronize when the user saves. That way, one view is locked and cannot be edited when the other is used for editing. This already introduces a new usability problem because the user can only edit one side at a time. The main issue, however, is that users can only save valid models. If the model is invalid in terms of the text, this means that there is a syntax error. Then it is impossible to derive the model from it that the diagram refers to. The other way around is similar. If the user saves the diagram, for example, in a state where there are multiple root objects, it will not be possible to save the textual model.

- **Bidirectional mapping.** Having two editors means that there are also two models. As an example we will use a model which represents nodes in the form of a tree. This example is visualized in Figure 6.1. A node class can have multiple children as indicated by the reflexive containment relationship. One possible textual representation of a node `A` containing nodes `B` and `C` is depicted on the left bottom side of Figure 6.1. In the diagram, this model will usually be visualized by representing each element in the form of a graphical node. Node `A` will be connected by directed connections with nodes `B` and `C` in the diagram. The semantics of the containment define that if a user deletes node `A`, also node `B` and `C` are deleted. This is, however, not the expected behavior in the diagram. Even if deleting the graphical node `A`, the user usually expects that `B` and `C` persist. This means that operations (e.g. the delete operation) cannot simply be mapped between the textual and graphical side. Operations such as create, delete, cut, paste, or drag and drop, thus, all have to be manually implemented in order to guarantee consistency.

While we do not want to state that these problems are unsolvable, it at least requires a lot of effort and thought to provide a correct and usable solution. All of the above challenges can immediately be solved by using a unidirectional mapping instead of a bidirectional one. This means that the underlying model can only be changed in the textual editor and the graphical representation is read-only. By using this approach developers can fully concentrate on creating good and usable graphical views, which aim at improving comprehension, instead of having to deal with the technical challenges named above.

As we have shown in Chapter 4, visualizations can come in very different forms and often graphical editing is not even possible or necessary. We, therefore, also use a unidirectional relationship between text and visualization.



```
node A {
  node B
  node C
}
```

```
Diagram {
  Node A
  Node B
  Node C
  Edge(A,B)
  Edge(A,C)
}
```

Figure 6.1: Metamodels of textual instance (left) and diagram instance (right) [16]

6.2 Usability and Aesthetics

As discussed in the previous section, diagram editor frameworks are not perfectly suited for creating visualizations due to multiple technical challenges that would have to be solved when graphical editing is involved. We would also like to stress that creating good visualizations also greatly relies on the base technology's capabilities of supporting usability and aesthetics. While programming languages can be seen as a means for communicating between computer and human, visualizations are solely for and used by humans. Usability and aesthetics are thus very important aspects developers should be able to focus on.

Bresciani et al. [6] tried to answer the question which factors foster business diagram adoption by concentrating on qualitative visualizations. They identify two dimensions that are also directly applicable to visualizations:

Perceived ease of use. Describes the "degree to which a person believes that using a particular system would be free of effort" [9]. The dimension can be subdivided into the following key factors.

- easy to understand
- easy to learn
- content categories relevance
- aesthetic value

Perceived usefulness. Describes the "degree to which a person believes that using a particular system would enhance his or her job performance" [9]. Key factors of this dimension include:

- allows to work faster
- improves job performances

- immediate insight
- adaptability to task

Especially aesthetic value and its effect on usability is often undervalued when developing visualizations. Cawthon et al. [7] investigate on the correlation between task abandonment, erroneous response time, and perceived aesthetic. The study's results are based on an online survey using 11 different data visualization techniques. The results illustrate that the visualizations that were perceived by the users as most aesthetic also perform relatively high in metrics of effectiveness, rate of task abandonment, and latency of erroneous response. Cawthon et al. , therefore, showed that aesthetics affect usability, which means that it should be considered when developing visualizations. Of course, aesthetic design requires a suitable rendering platform.

6.3 JavaFX

Introduction and Features

JavaFX is a software technology that allows developers to create rich cross-platform applications. JavaFX takes advantage of modern GPUs through hardware-accelerated graphics. It also provides well-designed programming interfaces which allow developers to combine graphics, animation, and UI controls. The goal of JavaFX is to be used across a wide variety of platforms and devices, such as smartphones, tablets, computers, desktops, embedded devices, and TVs [3, 12].

Before the introduction of JavaFX, rich-client applications were created by combining multiple separate libraries and APIs for media, UI controls, Web, 3D, and 2D APIs. As a graphical user interface toolkit, JavaFX tries to provide all of these capabilities out of the box. The language itself is compiled, statically typed, and declarative, and offers automatic data binding, triggers, animation, sequences, function types, inferred types, and error handling with Java-like exceptions. JavaFX lets developers access the complete Java API which allows to use existing Java libraries and tools. Oracle [23] names the following key features of JavaFX 8:

- **Java APIs.** JavaFX is a Java library. It consists of classes and interfaces which are written in Java.
- **FXML and Scene Builder.** UIs can be defined not only by using the APIs, but also by using FXML, which is an XML-based declarative markup language. It can also be considered to be a DSL and can be used by designers to create GUIs. The scene builder is a graphical editor for creating UIs and generates FXML markup.
- **WebView.** A component that uses WebKitHTML technology and makes it possible to embed web pages within JavaFX applications. Java-APIs can call JavaScript running in WebView and JavaScript can also call Java-APIs. HTML5 features such as Web Sockets, Web Workers, Web Fonts, and printing capabilities are supported as well.
- **Swing operability.** Swing applications can be updated with JavaFX features and Swing content can be embedded into JavaFX.

- **Built-in UI controls and CSS.** JavaFX supports all major UI controls including DatePicker and TreeTableView controls. Applications can be skinned using CSS.
- **3D graphics features.** Developers can use 3D geometry, cameras, and lights to create, display, and manipulate objects in 3D space.
- **Canvas API.** Allows drawing directly within an area of the JavaFX scene that consists of one graphical element.
- **Rich text support.** JavaFX supports bi-directional text and complex text scripts, such as Thai and Hindu in controls. It also supports multi-line and multi-style text in text nodes.
- **Multitouch support.** JavaFX supports multitouch gestures such as rotate, scroll, swipe, and zoom.
- **Hi-DPI support.** Hi-DPI displays are supported.
- **Hardware-accelerated graphics pipeline.** Graphics in JavaFX are based on the graphics rendering pipeline Prism. Prism is used for rendering graphics if the system uses a supported graphics card or GPU.
- **High-performance media engine.** The media pipeline facilitates the playback of web multimedia content. The media framework is based on the GStreamer multimedia framework.

This already indicates the power that comes with using a modern rendering platform. Almost all of the given features also have a great impact on visualizations for DSLs.

History

The language was initially created by Chris Oliver at a company called SeeBeyond and was known as F3 (Form Follows Function). Sun Microsystems acquired SeeBeyond in 2005 and unvealed F3 at the 2007 JavaOne conference as JavaFX. Sun Microsystems was later acquired by Oracle in 2009. In 2010, Oracle announced its plans to phase out the JavaFX scripting language and recreate the JavaFX platform for the Java platform as Java-based APIs. In October 2011 JavaFX 2.0 was released and Oracle announced its plans to take steps to open-source JavaFX to allow for an increase in adoption and faster bug fixes and enhancements. JavaFX 8 was released together with Java 8 in 2014. Table 6.1 shows a timeline of major JavaFX releases, including the features introduced in the respective versions [12].

Architecture

In the following we would like to provide a high level description of the JavaFX architecture and describe the most important architectural components. A layered diagram of the architecture is given in Figure 6.2. Although most of the shown components are not exposed publicly, knowing about them can improve users' understanding of how JavaFX applications work [23].

Release Date	Version	Platform	Description
December 4th 2008	1.0	Windows, MacOS	JavaFX Script Language, Production Suite, Media Playback
February 12th 2009	1.1	Windows, MacOS	New mobile development
June 2nd 2009	1.2	Windows, MacOS, Linux, Solaris	Skinnable UI controls, Charting API, and Performance improvements
April 22nd 2010	1.3	Windows, MacOS, Linux, Solaris	JavaFX Composer, TV Emulator, Mobile Emulator
October 3rd 2011	2.0	Windows	Rewritten for the Java Language
April 27, 2012	2.1	Windows, MacOS	The official version for the MacOS platform was released. Media supports H.264/MPEG-4 AVC. Supports a JavaScript bridge for web engine.
August 14, 2012	2.2	Windows, MacOS and Linux	The official version for the Linux platform was released. The following are the new APIs and tools: Canvas, HTTP Live Streaming, Touch events, Gestures, Image manipulation APIs and native application packaging.
March 18, 2014	8.0	Windows, MacOS and Linux	JavaFX 8 supports the following APIs: 3D graphics, Rich text support, Printing APIs and a JVM/JDK for embedded systems.

Table 6.1: Historical timeline of major JavaFX releases [12]

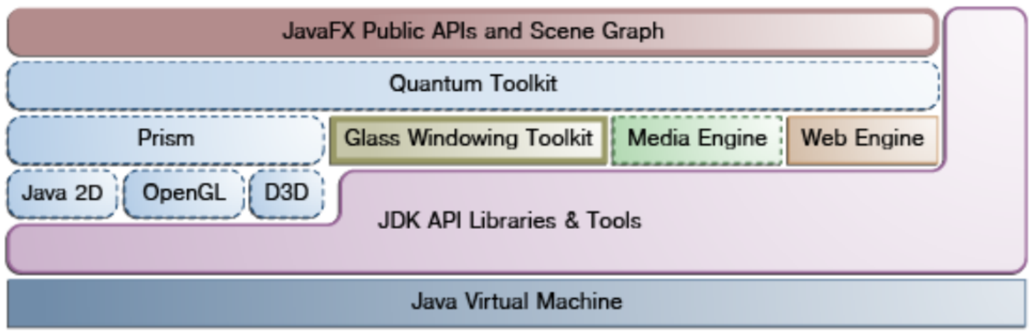


Figure 6.2: JavaFX architecture [23]

Scene Graph. The scene graph is a hierarchical tree of nodes that represents all visual elements of the created UI. It can handle input and can be rendered. Each node, except for the root node, in the scene graph has exactly one parent and zero or more children. A node has an ID, a style class, and a bounding volume and can also have effects (e.g. blurs and shadows), opacity, transforms, event handlers, and an application-specific state. The scene graph can directly include graphic primitives, text, controls, layout containers, images, and media. Animating elements in the scene graph can be accomplished by using the respective APIs.

JavaFX Public APIs. The top layer shown in Figure 6.2 also provides public APIs that support the development of rich client applications. The APIs allow the use of Java features such as generics, annotations, multithreading, and lambda expressions. They also facilitate the use of binding which includes high performance lazy binding, binding expressions, bound sequence expressions, and partial bind reevaluation. Furthermore, these APIs extend Java collections to include observable lists and maps, which enables the wiring of UI elements to data models.

Graphics System. The graphics system is comprised of the blue components in Figure 6.2. It supports 2D and 3D scene graphs and provides software rendering when hardware-accelerated rendering is not supported by the graphics hardware on a system. Prism processes rendering jobs and is responsible for rasterization and rendering JavaFX scenes. Depending on the device either DirectX 9, DirectX 11, OpenGL, or software rendering is used. Quantum Toolkit is comprised of Prism and Glass Windowing Toolkit and makes them available to the JavaFX layer above. Threading rules related to rendering versus events handling are managed by Quantum Toolkit as well.

Glass Windowing Toolkit. Glass Windowing Toolkit is the lowest level in the JavaFX graphics stack and responsible for providing native operating services, such as managing the windows, timers, and surfaces. It connects the JavaFX platform to the native operating system. It also manages the event queue by using the native operating system's event queue functionality to schedule thread usage. The system can run multiple threads including the JavaFX application thread, the Prism render thread, and a Media thread. We refer the interested reader to the official documentation for further information on these threads [23].

Media Engine. The Media Engine was designed for providing consistent behavior across platforms. It supports both visual and audio media in the form of MP3, AIFF, WAV, and FLV files. Media functionality is provided as three separate components, including the Media object, the MediaPlayer, and the MediaView. Live streaming is supported as well.

Web Engine. The Web Engine component is based on WebKit and provides a web viewer and full browsing functionality. WebKit is an open source web browser engine and supports HTML5, CSS, JavaScript, DOM, and SVG. Java calls can be controlled through JavaScript and vice versa.

6.4 Frameworks based on JavaFX

This section introduces two frameworks, namely FXDiagram and GEF4, which are based on JavaFX. Both of them were initially considered when selecting a technology for implementing visualizations for the practical part of this thesis.

FXDiagram

In 2011 Jan Koehnlein [1] made first attempts to implement unidirectional graphical views for textual representations of an object model. In a first prototypical framework called Generic Graph View, graphical views could be configured using two textual DSLs. One was used for mapping semantic objects to graphical elements, and the other for styling the graphical representation. The graphics were implemented using GEF and Zest layouts. Also in 2011, support for visualizing the syntax of a DSL grammar in Xtext was added. However, this visualization was unrelated to Generic Graph View. The graphical view supported navigation between the editor and the view. The diagram was implemented in Draw2d using a custom layout algorithm.

In 2012 Koehnlein added diagram discovery support to Generic Graph View which allowed users to step-by-step extend the diagram content starting from an initial node and focus on a specific part of the model. Hovering over nodes which have connections to other nodes that are not yet shown resulted in buttons appearing around that node. The buttons could then be used to incrementally add new neighbor nodes and connections. In order to improve the user experience during selecting potential neighbor nodes Koehnlein later added multitouch gestures to the framework. GEF, by default, does not support multitouch gestures. Therefore, Koehnlein extended multiple GEF classes in order to add behavior for implementing zoom, swipe, and rotate gestures.

Due to rising expectations and the many insufficiencies of GEF3, Koehnlein started using JavaFX as a base technology in 2013. Using JavaFX and Xtend, Koehnlein started developing a framework, which was later released as FXDiagram [2].

FXDiagram adds a thin abstraction layer on top of JavaFX. The framework only defines abstractions for nodes and edges in order to solve hard issues like edge routing and autolayout. It, thus, allows developers to leverage the full power of JavaFX. FXDiagram has already addressed multiple challenges and supports the following features, which can be reused when implementing visualizations.

- **Moving nodes.** In FXDiagram nodes can be fluently moved and edges are automatically rerouted while doing so. This behavior is opposed to other frameworks, where edges are not changed during the moving operation but rerouted after dropping the node at the final position.

- **Edges.** FXDiagram supports cubic Bézier splines, quadratic spline and polylines for connecting nodes. Off-by-one rendering errors are avoided by relying on double-coordinates.

[1] `koehnlein.blogspot.co.at`

[2] `jankoehnlein.github.io/FXDiagram`

Edge labels are aligned in parallel to the corresponding edges in order to avoid overlapping elements.

- **Autolayout.** FXDiagram includes advanced autolayout functionality based on KIELER[3] and Graphviz[4].
- **Infinite canvas.** Diagrams can be viewed on an infinite canvas.
- **Diagram nesting.** Nodes can contain other diagrams. FXDiagram provides two ways of nesting diagrams. When using level of detail nesting the inner diagram is revealed after the user zoomed into the container node far enough. When using the second nesting approach, the inner diagram is opened using a transition animation when the user double-clicks on the container node.
- **Diagram discovery.** FXDiagram handles very large diagrams by displaying only specific points of interest and letting the user discover other parts of the diagram if requested.
- **Diagram repair.** FXDiagram avoids using automatic synchronization when the model is changed. Changed and repairable elements are highlighted by adding a blinking animation to nodes, unrestorable elements are made transparent. The user can then manually repair the diagram.
- **Undo and redo.** This feature allows browsing the diagram changes over time. FXDiagram restores the view-ports and uses animated transitions in order to improve usability.
- **Xtext integration.** FXDiagram is integrated well with Xtext and provides integration between the graphical view and the textual editor.
- **Gestures and transformations.** FXDiagram supports multitouch gestures including zooming, rotating, and scrolling.

Besides providing many features out of the box, one of FXDiagram's main goals is to improve the user experience. This includes the use of modern visuals, real-life metaphors, and short transitions that guide the user's eye. FXDiagram also avoids dialogs, property panes, and toolbars in order to use the maximum amount of screen space for the diagram. A graphical menu is shown only when the user needs it. Metaphors are an important means of improving usability and are extensively used in FXDiagram. When selecting a node, the node's size is increased and a shadow is added. This is a metaphor for lifting up something from the desktop. Control points of Bézier curves may be hard to understand for users. FXDiagram uses the metaphor of magnetism for this, and renders control points as magnets. Graphical choosers are used for selecting neighbor nodes in diagram discovery and alignment lines are shown for arranging nodes.

[3]`rtsys.informatik.uni-kiel.de/en/research/kieler`
[4]`graphviz.org`

GEF4

The new version of GEF has been a vision since 2010 when developers started to revise the Zest API, which has been backwards compatible since 2007. At that time Draw2d and JFace were still used as underlying rendering technologies. The developers realized that many issues could not be resolved without breaking the API. In 2011 Nyssen [5] initiated the renewal of the Draw2d and GEF (MVC) 3 components, whose API have been kept stable since 2004. The new double-based geometry API was later finalized in 2012. At that time multiple GEF components were independently developed by different engineers. In order to improve the development process, developers decided to unite and work on the new generation of GEF under the name GEF4. GEF4 was initially developed without dependencies to Draw2d, GEF (MVC), and Zest, in order not to affect any adopters. As a first step the revised Zest component was migrated to the GEF4 namespace, and the new geometry component was adopted. After that, the replacing of Draw2d and GEF (MVC) 3, using JavaFX instead of SWT as a rendering platform, was initiated.

In June 2015 the first GEF4 components were released with the Mars release of Eclipse. The components offered a comparable functionality to Draw2d and GEF (MVC) while using JavaFX for rendering purposes. In June 2016 the GEF4 components were released in version 1.0.0. The release included a cleaned up API, improvements in connection handling, and an improved MVC component.

GEF4 includes the following features:

- **Geometric abstractions.** GEF4 offers different abstractions (euclidean, projective, planar), to perform double-based geometric calculations in two-dimensional space. Beside many other basic planar abstractions GEF4 includes Bézier curves of arbitrary degree, poly-Béziers, curved polygons, quadratic and cubic curves, and polylines. Furthermore, visual anchors and connection abstractions are provided.

- **Live feedback.** Similar to FXDiagram, the user receives live feedback, e.g. in the form of edges being automatically rerouted while moving nodes.

- **Autolayout.** GEF4 provides abstractions to integrate layout algorithms. It is intended to align the provided abstractions in order to also use layout algorithms provided by the KIELER framework.

- **Diagram nesting.** GEF4 includes nesting of diagrams and includes a semantic zoom feature.

- **Editors and views.** GEF4 allows developers to create both graphical editors and views. Editors and views can be deployed as part of standalone Java applications and may also be deployed in the web.

- **Infinite canvas.** As opposed to previous versions, GEF4 uses an infinite canvas approach.

- **Gestures and transformations.** GEF4 supports multitouch gestures and offers rotations and other affine transformations.

[5] `nyssen.blogspot.co.at`

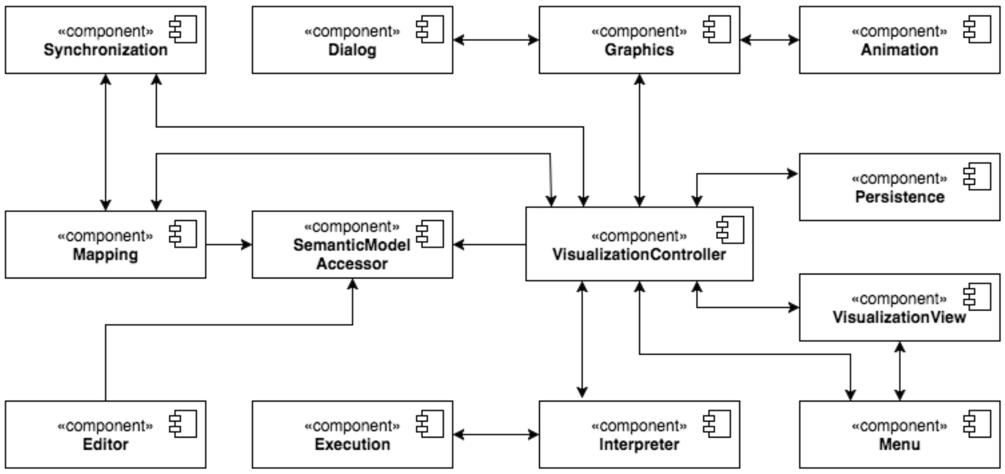


Figure 6.3: Visualization architecture

- **Undo and redo.** This feature was also available in prior versions to GEF4.

The shown features are similar to the ones provided by FXDiagram. Both frameworks could, thus, be used for implementing visualizations and one is not clearly superior to the other. The fact that GEF4 is mainly still focused on developing graphical editors and that usability features are less sophisticated than in FXDiagram, lead to the decision of preferring FXDiagram over GEF4 in the practical part of this thesis. A detailed comparison of the two frameworks is beyond the scope of this thesis and reveals potential for future work.

6.5 Conceptional Architecture

This section introduces a technology-independent architecture showing the main components of a visualization. The components and their interactions are depicted in Figure 6.3. Please note that the actual architecture may vary depending on the framework used. Different frameworks may apply different architectural styles. However, most of the shown components are usually required for visualizations and included in all architectures in some form. Interaction types, e.g. procedure calls or events, may also vary depending on the concrete architecture and are not specifically indicated in Figure 6.3. In the following the components are described in more detail. References to the respective concepts of the visualizations implemented for Use Case 1 are given if appropriate.

Editor. Users usually require means to modify the Semantic Model in some form. The Editor component allows the user to view and edit the Semantic Model. As described in Section 3.1, there are various types of Concrete Syntaxes of DSLs that can be used for interacting with the underlying model. The Semantic Model can, of course, also be directly modified without the need for an additional CS. The Editor component should offer a view and means for conveniently and effectively interacting with the DSL.

In Use Case 1 visualizations were developed for languages, which were created using Xtext. Xtext provides a textual editor for created languages out of the box. The editor can be used within IDEA, Eclipse, and all common web browsers. It supports features such as syntax coloring, semantic coloring, error checking, auto-completion, automatic formatting, and rename refactoring.

VisualizationView. Besides the view provided by the Editor, also a view for showing the visualizations is required. The VisualizationView component should offer means for viewing visualizations and interacting with them. The view itself should provide a canvas for displaying graphical elements. Users should be able to navigate inside the canvas by mouse and gestures. The respective handlers are managed by the VisualizationView component and events are forwarded to other components if required.

FXDiagram, which was used in Use Case 1, uses an SWT-based Eclipse View within Eclipse. JavaFX-content is embedded into the view by using FXCanvas, which bridges SWT and JavaFX. Mouse and gesture events are converted respectively and can be handled by JavaFX event handlers. FXDiagram offers functionality for navigating within the canvas and moving graphical elements by drag and drop, as well as scroll and zoom gestures. The view uses tabs for managing multiple open visualizations at a time.

Menu. The user interface often offers menus for managing visualizations and accessing features. The Menu component manages general and visualization-specific menus. A general menu is necessary for letting the user decide which visualization to show. Visualization-specific menus may be used for letting the user access operations and features related to the currently shown visualization.

Using Xtext and FXDiagram, visualizations can be opened by accessing a context menu on textual elements within the editor. The general menu shows all visualizations that can be opened for the selected element. Selecting a menu entry opens the graphical view or a new tab, if a visualization is already shown. A visualization-specific graphical context menu can be opened by right-clicking in the graphical view. Actions such as zooming, autolayout, or opening new visualizations that are based on the current visualization, can be accessed through this menu.

SemanticModelAccessor. The Editor component needs to access the Semantic Model for viewing and editing its contents. Furthermore, the components responsible for creating visualizations need to access the Semantic Model as well. These components may, of course, not edit the Semantic Model as visualizations are read only. The SemanticModelAccessor represents a component for reading and editing the elements of the Semantic Model. The component is used by the Editor as well as visualization and mapping components.

FXDiagram uses `DomainObjectDescriptors` for accessing domain objects. A method allows reading domain objects safely by opening an appropriate transaction, executing a given lambda, and finally closing the transaction and returning the result. The descriptors are used to avoid having hard references to EMF objects, and for persisting domain objects when storing diagrams.

VisualizationController. The VisualizationController is the central component for creating and managing visualizations as well as propagating actions on a visualization. It is thus accessed by components that are responsible for tasks related to visualization creation, synchronization, mapping, and persistence.

In the implementation of Use Case 1 each visualization has a separate controller which is linked with the visualization itself by means of FXDiagram's `XDiagram` class which is used for initializing diagrams and managing nodes and connections. This enables initiating the interpretation and graphical construction of the visualization as well as the registration of menu actions.

Interpreter. Often, the visualization cannot be created by simply mapping the Semantic Model's elements to graphical elements. The Interpreter component is used to analyze the Semantic Model with respect to a specific aspect. After interpreting the model, the creation of the visualization can be initiated. It is also possible to generate a textual report based on the data obtained from the Interpreter. In fact, one of this thesis' research questions aims at investigating if transforming the Interpreter's output to a graphical visualization is necessary in order to improve comprehension, or if a direct textual output of the Interpreter is sufficient.

In Use Case 1, the model has to be interpreted, for example, in order to find a valid path for the traversal visualizations. The Interpreter transforms the room's model into a graph and then applies Dijkstra's algorithm for finding the shortest path from the entry to the exit. Interpretation is also necessary prior to creating the chart visualizations. We refer the reader to Chapter 4 for detailed descriptions of the visualizations.

Execution. Models can not only be interpreted but may also be executed [19]. Models can be made executable by using code generators or by using model interpreters. Model execution is the basis for debugging, which allows step-wise execution, setting breakpoints, and inspecting variables. The visualization and animation of an execution can support the user and may increase comprehension of the designed system. The Execution component generates a sequence of actions that can be visualized and corresponds to the execution of the model.

In Use Case 1 the Execution component is used to generate player actions for simulating the level traversal. The actions are then visualized by transforming actions to animations.

Graphics. The Graphics component is used for generating static visualizations consisting of graphical nodes and edges. By directly accessing the underlying media and graphics technology, it should be possible to include images, video, audio, controls, as well as arbitrarily complex geometric 2D and 3D shapes.

FXDiagram allows developers to use JavaFX for creating a diagram's graphical elements. Therefore, all of the features described in Section 6.3 can be used to create visualizations. Visualizations in Use Case 1, use the Graphics component for creating basic nodes and edges, images, layouts, UI controls, charts, basic 3D shapes, and 3D models.

Animation. Animations are very important and should be used in order to guide the user's eye during all operations. Animations should also be extensively used in dynamic visualizations required for visualizing executions and simulations. The Animation component is responsible for creating these animations and adding dynamic images on top of static visualizations.

FXDiagram uses animations when executing operations such as zoom, autolayout, undo-redo, opening nested diagrams, or opening the graphical menu. The visualizations of Use Case 1 use sprite animations for visualizing walk and fight cycles and transition animations in parallel which are all created by the Animation component.

Dialog. Visualizations can and should be interactive. Users may need to declare some information before a visualization can be created. The Dialog component is used for creating arbitrary dialogs or forms which can be shown in the VisualizationView by themselves, or inserted in visualizations.

In Use Case 1 the Dialog component is, for example, used for configuring the level traversal simulation. Before the visualization is shown and the simulation is started, the user has to enter the players properties, as well as the simulation speed. The Dialog component also inserts an information dialog showing the player's health points during the traversal.

Mapping. Elements shown in the editor view should be linked with their graphical representation in the visualization if possible. As visualizations should improve the comprehension of textual models, it may be beneficial for developers to highlight an element in one view if the corresponding element is selected in the other view. The Mapping component manages the links between elements of the Semantic Model and graphical elements such as nodes and edges.

Mappings are usually managed by FXDiagram and can be defined using the diagram configuration. The default implementation allows to highlight elements in the editor by double-clicking on graphical elements. Graphical elements can be revealed by using the context menu on textual elements in the editor. All visualizations of Use Case 1 were created by using the default mapping behavior. Custom mapping implementations were required in Use Case 2, which is introduced in Chapter 8. Because graphical elements which could not be mapped to any element of the Semantic Model were visualized, the diagram configuration could not be used for specifying the mappings. In such cases, custom implementations are required.

Synchronization. Using a unidirectional relation between the editor and graphical view greatly simplifies the synchronization between the two model representations. However, the view has to be synchronized at some point after the developers changed the underlying model in the editor. This is handled by the Synchronization component. Depending on the implementation, the graphical view can be synchronized automatically after performing changes in the editor, after saving, or manually by explicitly requesting the graphical view to be synchronized. The user should receive some kind of graphical feedback about how changes in the editor affect the graphical view.

In FXDiagram synchronization is based on the mappings defined in the diagram configuration and mostly handled by the framework out of the box. Changed and repairable elements are highlighted by adding a blinking animation to nodes. Unrestorable elements are made transparent. After manually requesting a synchronization, repairable elements are restored and unrestorable elements removed.

Persistence. Depending on the framework and visualization, it might be possible for users to edit parts of the visualization. It is important to note that changes to the visualization do, however, not change the Semantic Model. For example, the position of graphical nodes may be changed without affecting the underlying model. It may also be possible to delete elements in order to view only specific parts of the visualization. Again, these changes are not propagated to the model. After having edited a visualization, users may want to save the current state. The Persistence component is responsible for storing and loading visualizations.

FXDiagram does not have a separate diagram model, but serializes the JavaFX scenegraph directly in JSON notation. Using an active annotation developers can define which properties of a class, e.g. representing a node, are important for saving. Some properties may be recoverable from the domain object and do not have to be stored.

This section presented the main architectural components that may be used when creating visualizations. As has been mentioned previously, the components should illustrate a visualization from a conceptional point of view and should provide a basic insight into which components are usually required. The architecture is technology-independent and components can be implemented using an arbitrary language and framework.

The next chapter addresses implementation specific details, which are relevant when creating visualizations with the technologies used in the practical part.

CHAPTER 7

Implementation

This chapter presents details on the implementation of visualizations. We first define the technologies used besides FXDiagram, namely Xtext and Xtend. After that, we show how some of the components described in the previous chapter can be implemented and how developers can utilize FXDiagram's features. Finally, we present some of the challenges we discovered during implementing the first use case. Most of these challenges are relevant when developing visualizations in general, and do not only occur when using FXDiagram. The implementation of Use Case 1 is available on GitHub [1].

7.1 Technological Background

Xtext

Xtext [2] is a framework for the development of programming languages and DSLs. Languages are defined using a dedicated grammar language. Starting from the defined grammar, a full language infrastructure including a parser, linker, typechecker, compiler, and editing support for Eclipse, IntelliJ IDEA and all common browsers is generated.

The grammar language consists of a collection of parser rules which define both the language's textual concrete syntax and its mapping to elements of the metamodel. The language used in Use Case 1 was implemented using Xtext. A part of the language's grammar, including the parser rules `Level` and `Room`, is given in Listing 7.1. A parser rule begins with a name and a colon, and is followed by the rule body. Each parser rule creates a corresponding class in the metamodel. By default the parser rule's name defines the name of the respective metaclass. This means that objects in the Semantic Model are of a type that is defined by means of production rules. For example, the parser rule `Room` creates a metaclass *Room* in the metamodel. Instances of rooms, e.g. 'Room 1', 'Room 2', are represented in the Semantic Model as separate objects of

[1] `github.com/alealt/ldl-visualization`
[2] `eclipse.org/Xtext`

type *Room*. The first parser rule in every grammar defines where the parser starts. It also defines the root element of the metamodel. String literals, which in Xtext can be expressed with either single or double quotes, define keywords, i.e. parts of the DSL's CS.

Attributes of metaclasses can be defined using three different assignment operators. The simple = operator is used for attributes taking only one element. The += operator is used for defining attributes of a list type. Finally, the ?= operator results in a boolean attribute, which is set to true if the right side of the assignment was consumed. The symbol on the right side of the assignment can specify one of four possible cardinalities. The possibilities are exactly one (no operator), zero or one (operator ?), zero or more (operator *), and one or more (operator +). The right hand side of an assignment can also be a cross-reference, which is indicated by a rule name in square brackets. The rule name, however, does not refer to another rule but to a metaclass.

```
Level:
  'spawn' '=>' spawnRoom=[Room]
  'goal' '=>' goalRoom=[Room]

  rooms+=Room+
;

Room:
  'room' name=ID '{'
    'columns' '='? columns=INT
    'rows' '='? rows=INT

    entry = Entry
    exit = Exit exitConnection = Connection?

    walls += Wall*
    trapdoors += Trapdoor*
    monsters += Monster*

  '}'
;
...
```

Listing 7.1: LDL grammar

Xtext internally relies on the Eclipse Modeling Framework (EMF) and uses EMF models as the in-memory representation, i.e. the Semantic Model. Ecore is a variant of the EMOF standard and acts as EMF's meta-metamodel. Ecore models, thus, are used for representing a language's metamodel. Instances of the metamodel correspond to the Semantic Model. The metamodel defines the types of the Semantic Model as `EClasses`, which can have `EAttributes`. The Semantic Model consists of `EObjects`. This model is created by the parser when processing a textual input file.

Xtend

Xtend [3] is a dialect of Java which compiles into readable Java source code. Xtend aims at having a more concise and readable syntax than Java and provides features such as type inference, extension methods, lambda expressions, and multi-line template expressions. Xtend is completely interoperable with Java, which means that all Java libraries can be reused.

The language itself is implemented in Xtext and is a proof of concept of how involved a language implemented using Xtext can be [4]. Xtend is statically typed and uses the Java type

[3] `eclipse.org/Xtend`

system. Most of the language concepts are similar to Java. However, Xtend aims at avoiding redundant linguistic features that make programs verbose. For example, semicolons are optional, and methods are public by default. Furthermore, variables are declared using `var` or `val` and their type is inferred. Xtend also provides some syntactic sugar for getter- and setter-methods. For example, instead of writing `p.setName("...")` developers can simply write `p.name = "..."`. Furthermore, parenthesis are optional when invocating methods without parameters.

One of Xtend's features is especially useful for the initialization of objects. Objects representing layout containers, or graphical shapes often have many parameters used for customization. Builder classes were introduced for simplifying the initialization of such objects. The implementation, however, was based on two bugs in JDK6 and JDK7 [4]. The bugs were fixed in JDK8 which lead to Builders not working anymore in later versions. The code usually used for the initialization of a rectangle in JavaFX 8 is shown in Listing 7.2. There are, obviously, many redundant parts. The only way of writing this code more concisely is by using constructors with many parameters. In the given example, the use of the constructor does, however, worsen the readability.

```
Rectangle rectangle = new Rectangle(50,30, Color.BLUE);
rectangle.setStroke(Color.RED);
rectangle.setStrokeWidth(1.5);
rectangle.setArcWidth(15);
rectangle.setArcHeight(12);
```

Listing 7.2: Rectangle creation in Java

Xtend's with operator is very useful for improving the readability of such initialization. The with operator (=>) passes the left hand side argument to the lambda on the right side and returns the left hand after that. This enables to write very readable initialization code such as shown in Listing 7.3. The with operator also facilitates the creation of object trees. This is very useful as JavaFX' scene graph consists of a hierarchical tree of nodes.

```
val rectangle = new Rectangle => [
  width = 50
  height = 30
  fill = Color::BLUE
  stroke = Color::RED
  strokeWidth = 1.5
  arcWidth = 15
  arcHeight = 12
]
```

Listing 7.3: Rectangle creation in Xtend

FXDiagram is almost entirely written in Xtend and also promotes the use of Xtend when creating diagram views. Therefore, some of the code snippets in the following sections are written in Xtend as well.

7.2 Details

This section provides implementation specific information gained during this thesis' practical part. It is also demonstrated how a framework for creating visualizations could implement the most important components described in the previous chapter.

[4] `mail.openjdk.java.net/pipermail/openjfx-dev/2013-March/006725.html`

Diagram Configuration

FXDiagram provides a high-level mapping API [5] which can be used for creating diagram views for models. The mapping configuration can be registered using an extension point, which is offered by both the Eclipse and IDEA [6] integration. A diagram configuration serves three purposes:

1. Define how domain objects are mapped to nodes, edges, and diagrams.
2. Define on which domain objects a "Show in FXDiagram" action is available.
3. Define how the domain objects can be accessed and serialized.

Diagram configurations can be implemented in Java or Xtend. The structure of a diagram configuration for a state machine is given in Listing 7.4. FXDiagram also provides abstract superclasses for Xtext-based models, Eclipse-based models, and IDEA's PSI models.

```
class StatemachineDiagramConfig extends AbstractDiagramConfig {
  // fields to define mappings (1)
  val stateNode = new NodeMapping<State>...
  val stateLabel = new NodeHeadingMapping<State>...
  val transitionConnection = new ConnectionMapping<Transition>...
  val eventLabel = new ConnectionLabelMapping<Event>...

  // method to define entry points (2)
  override protected <ARG> entryCalls...

  // method defining the domain object access (3)
  override protected createDomainObjectProvider...
}
```

Listing 7.4: FXDiagram diagram configuration structure (taken from the FXDiagram website)

Mappings are implemented as fields in the diagram configuration class and are used to map domain objects to diagram elements. Domain objects can be mapped to nodes, connections, labels, and diagrams. Mappings allow the developer to specify what else to create when a mapping is executed (e.g. connections when a node is created). Mappings can also be used to specify that a custom implementation of a node, connection, or diagram should be used. This enables developers to create arbitrary diagram elements utilizing JavaFX. Listing 7.5 shows a mapping definition for the 2D room diagram used in Use Case 1.

```
val roomDiagram = new DiagramMapping<Room>(this, 'roomDiagram', 'RoomDiagram') {
  override XDiagram createDiagram(IMappedElementDescriptor<Room> descriptor) {
    new RoomDiagram(descriptor)
  }

  override calls() {
    // when adding a room diagram also add a node for monster, wall, trapdoor, ...
    monsterNode.nodeForEach[monsters]
    wallNode.nodeForEach[walls]
    trapdoorNode.nodeForEach[trapdoors]
    entryNode.nodeFor[entry]
    exitNode.nodeFor[exit]
  }
}
```

Listing 7.5: Mapping definition

[5] http://jankoehnlein.github.io/FXDiagram/hla
[6] jetbrains.com/idea

Entry points are used to define on which kind of language elements the user can execute which mappings. Listing 7.6 shows a part of the entry point definition of the diagram definition used in Use Case 1. Users can execute a “Show in FXDiagram as” action on elements of type `Level` and `Room` by right clicking on these elements in the editor. When selecting an element of type `Level`, users can execute the `levelDiagram` mapping which opens the level overview diagram in which rooms are represented as simple nodes. When selecting an element of type `Room`, users can choose between three mappings: The level overview diagram, the 2D room visualization, and the 3D visualization. References to the respective domain objects are passed to the mapping, and can then be accessed when initializing custom diagrams and nodes.

```
override protected <ARG> entryCalls(ARG domainArgument, extension MappingAcceptor<ARG>
     acceptor) {
  switch domainArgument {
    Level:
      add(levelDiagram, [domainArgument.getContainerOfType(Level)])
    Room: {
      add(levelDiagram, [domainArgument.getContainerOfType(Level)])
      add(roomDiagram, [domainArgument])
      add(room3DDiagram, [domainArgument])
    }
    ....
  }
}
```

Listing 7.6: Entry point definition

The access and serialization of domain objects is defined by means of domain object providers. This is predefined for developers when using abstract diagram configuration classes, e.g. for Xtext-based models.

After having defined the diagram configuration developers have to specify how the runtime infrastructure of the IDE can pick up the configuration. Using Xtext and Eclipse, configurations are registered to the extension point in the plugin.xml or fragment.xml in the ui project. This is demonstrated in Listing 7.7.

```
<extension point=de.fxdiagram.mapping.fxDiagramConfig">
  <config
    id="at.ac.tuwien.big.leveldesign.LevelDiagramConfig"
    label="Level"
    class="at.ac.tuwien.big.leveldesign.ui.LevelDesignExecutableExtensionFactory:
        at.ac.tuwien.big.leveldesign.ui.config.LevelDiagramConfig">
  </config>
</extension>
```

Listing 7.7: Registration of diagram configuration

External Interpretation

As described in the previous chapter, the Interpreter component is used to analyze the Semantic Model with respect to a specific aspect prior to creating the visualization. The Interpreter, however, does not necessarily have to be implemented in the language used for creating the visualizations. In the second use case, which is introduced in Chapter 8, the main interpretation was done using Prolog. Prolog is a declarative language and a program’s logic is expressed in terms of relations. A computation is initiated by running queries over the relations. The Prolog-Interpreter was integrated using Projog [7], which is an implementation of the Prolog language for the Java platform, and SWI-Prolog [8], a Prolog environment. The Interpreter component can, thus, be constructed using the frameworks and languages best suited for interpreting the underlying model.

7.3 Challenges

This section discusses some of the particularities and challenges of implementing visualizations. The challenges were identified while working with the technologies used in the practical part. Developers, however, face most of these challenges irrespective of the technology they use. The solutions may, of course, vary depending on the technologies used.

Mapping Non-Existing Semantic Elements

As illustrated in Section 7.2, using the high-level mapping API provided by FXDiagram, graphical elements can be conveniently mapped to elements of the Semantic Model. A visualization framework should provide a mapping concept similar to this, in order to facilitate the fast development of visualizations. By doing so the framework can provide default implementations for synchronization and interactions between the editor view and the visualization view (e.g. selecting an element in one view, and highlighting the corresponding element in the other view).

While developing the visualizations for the second use case, we experienced a common situation which could not be easily handled by the framework used. It may be necessary to view graphical elements, e.g. nodes or connections, which can not be uniquely mapped to an element of the Semantic Model. This was the case in Use Case 2 as a model for describing a multi-dimensional state machine consisted of a sequence of nested rules which describe when and how state transitions can occur. One visualization, which basically is a state machine diagram, aims at making these implicit state transitions explicit. Nodes representing the overall state of the system, however, do not have a corresponding element in the Semantic Model. Therefore, the high-level mapping API provided by FXDiagram could not be used for defining these mappings. While it is very easy to manually add nodes and edges to a diagram in FXDiagram, manually adding functionality for the interactions and synchronization between the textual and graphical element is a bit more cumbersome.

Another common situation which also hindered the usage of the mapping API in the use case given above, are multiplicities between graphical elements and semantic elements different than 1:1. The framework assumes that one graphical element, always has exactly one corresponding element in the textual editor and, therefore, also in the Semantic Model. However, in the example given above it is possible that one rule can yield multiple transitions in the state machine diagram. It might even be possible (although this was not the case in Use Case 2), that a transition is yielded by multiple rules. Visualizations may, therefore, require 1:1, 1:n, and even n:m relations between graphical elements and semantic elements.

[7] `projog.org`
[8] `swi-prolog.org`

A final interesting challenge with respect to mappings are languages which make use of Xtext's cross-references. Intuitively a cross-reference may be visualized by means of a connection, which starts at the element in which the cross-reference is defined and ends at the referenced element. To the best of our knowledge, this is also not possible using the high-level mapping API. This is due to the fact that a graphical element, e.g. the connection, again has to be mapped to an element of the Semantic Model, which in the case of a cross-reference does not exist. Connections can, of course, be added to the diagram manually. Interaction behavior has to be implemented manually, however, as well.

In the following we present our solutions for adding interaction behavior to elements which were not added to diagrams using the mapping API.

The first direction — from the graphical view to the editor — can simply be added by using the functionality already provided by FXDiagram. The code shown in Listing 7.8 shows the construction of a connection for a given transition `t`. Besides setting the source and target nodes of the connection the handling of the click event is important. By accessing the domain object provider, the editor can be accessed. The method `getCachedEditor` accesses the Xtext editor and selects the text region corresponding to the given transition's `XtextEObjectID` which uniquely identifies the semantic element. Using FXDiagram, it is thus very easy to select arbitrary elements in the textual view by using the respective IDs. We refer the interested reader to the source code of FXDiagram for details on the implementation of `getCachedEditor`.

```
var conn = new XConnection => [
  source = t.source
  target = t.target
  onMouseClicked = [it |
    if (it.getClickCount() ==1) {
      root.getDomainObjectProvider(XtextDomainObjectProvider)
        .getCachedEditor(t.xtextEObjectID, true, true)
    }
  ]
]
```

Listing 7.8: Interaction from view to editor

The second direction — from the editor to the graphical view — is slightly more involved and is based on the interaction implementations of FXDiagram. We would like to reveal the graphical elements by selecting a context menu entry of the corresponding textual element. The Eclipse IDE uses commands to contribute actions to the user interface. There are three extension points which are relevant when adding a context menu and the respective actions to the editor.

- **org.eclipse.ui.commands.** A command is a declarative description of a component without implementation details.
- **org.eclipse.ui.handlers.** The handler defines the behavior. This is done by referencing the Java class, which contains the behavior implementation.
- **org.eclipse.ui.menus.** This extension point is used to define where the command should be included in the user interface.

A code snippet showing the relevant configuration of these three extension points in the plugin.xml is given in Listing 7.9. The command name specifies the name of the context menu entry. The menu's extension point specifies that the entry is shown in context menus in the text editor. The position of the menu entry is defined by `before=group.edit`. Finally, the behavior when selecting the context menu entry is defined in the class `ShowInViewHandler`.

The `ShowInViewHandler` extends `AbstractHandler` and implements the method `execute`. The most important parts of the implementation, without error handling, are shown in Listing 7.10. Starting from the selected text the corresponding selected EObject is computed. After that, the active graphical view and its current diagram is located. Finally, the method for highlighting the respective elements in the diagram is called.

```
<extension point="org.eclipse.ui.commands">
  <command name="Reveal Rule In Diagram"
    id="org.xtext.example.mydsl.ui.handler.showInViewCommand">
  </command>
</extension>

<extension point="org.eclipse.ui.handlers">
  <handler class="org.xtext.example.mydsl.ui.MDML04ExecutableExtensionFactory:
    org.xtext.example.mydsl.ui.overallstate.ShowInViewHandler"
    commandId="org.xtext.example.mydsl.ui.handler.showInViewCommand">
  </handler>
</extension>

<extension point="org.eclipse.ui.menus">
  <menuContribution locationURI="popup:#TextEditorContext?before=group.edit">
    <command commandId="org.xtext.example.mydsl.ui.handler.showInViewCommand">
    </command>
  </menuContribution>
</extension>
```

Listing 7.9: Context menu configuration

```
override Object execute(ExecutionEvent event) {
  val editor = EditorUtils.getActiveXtextEditor(event)
  val selection = editor.selectionProvider.selection as ITextSelection
  editor.document.readOnly [
    var selectedElement = eObjectAtOffsetHelper.resolveElementAt(it, selection.offset)
    val page = PlatformUI?.workbench?.activeWorkbenchWindow?.activePage
    val activePart = page?.activePart
    val view = activePart.site.page.showView("de.fxdiagram.eclipse.FXDiagramView")
    ((view as FXDiagramView).currentRoot.diagram).highlightElement(selectedElement)
  ]
}
```

Listing 7.10: Handler implementation

Node Positioning in 2D Space

Frameworks used for creating graphical editors and views often allow the user to move and arrange graphical elements on the canvas. The autolayout feature also re-positions elements in order to find a layout which presents the graphical contents as clear as possible. FXDiagram, for example, by default supports the fluent movement of nodes and connections as well as rearranging elements by using autolayout. This, however, is not possible if the position of a graphical element on the canvas represents semantic information in the underlying model. For example, the visualization of a room in Use Case 1 heavily relies on correctly positioning elements on the canvas. Monsters, trapdoors, entries, and exits have to be positioned with respect to the coordinates defined in the Semantic Model. Walls are constructed and positioned by interpreting coordinates correctly. Moving or automatically arranging these elements would, therefore, result in a change of the underlying model and has to be prevented.

FXDiagram implements behavior by means of dedicated classes. Any kind of user interoperability or liveness of a graphical shape is encompassed in such a behavior. This allows to compose multiple behaviors instead of implementing everything in the shapes. Avoiding the movement of nodes can, therefore, simply be achieved by removing the move behavior from nodes, which are positioned based on semantic information.

3D Visualizations and 3D Modeling

JavaFX offers interesting and easy to use 3D graphics features. The creation of three dimensional scenes using basic polygons such as boxes and spheres can easily be achieved by using well designed APIs. Introducing cameras, adjusting perspectives and introducing lighting is possible as well. JavaFX' capabilities of creating 3D content is also very interesting for visualizations. We, therefore, also tried implementing a 3D visualization in the first use case. This immediately yielded the question of in how far FXDiagram can be used for visualizing 3D content. In the following we present our findings.

FXDiagram is definitely a great framework for implementing two dimensional diagrams. However, at the time of writing it seems that FXDiagram was not designed with the aim of supporting the creation of 3D diagrams. We, therefore, identified some of the areas that are affected by changes when extending a 2D visualization framework in order to support 3D content.

- **Navigation.** Navigating in a 3D scene is very different than in 2D. In a 2D diagram it is sufficient to support exploring the canvas by two scrollbars or drag-and-drop gestures. Exploring a scene in 3D requires gestures for navigating along all three axes, as well as changing the user's perspective on the scene.

- **Interaction.** The interaction with and transformation of graphical elements is also more complex in 3D scenes. Usually nodes are moved by drag-and-drop gestures in 2D. This is, however, not sufficient for moving nodes to an arbitrary position in 3D. Implementing intuitive ways for modifying connections may is also be more difficult in 3D. The same holds for transformations such as rotation or scaling. Furthermore, the integration between editor and view may need to be revised.

- **Autolayout.** Usually layout algorithms have to be changed for supporting the autolayout feature in 3D. Automatic edge routing may also need to be changed.

- **Animations.** Animations used by the framework for improving the user experience may need to be extended. FXDiagram extensively uses animations for guiding the user's eye, for example, during undo and redo, when opening nested nodes, or during autolayouting.

Most of these areas are deeply rooted in the case of FXDiagram. Therefore, it appears not to be possible to create 3D visualizations without loosing any of the framework's main features.

However, JavaFX provides a class named `SubScene` for embedding 3D content into a 2D scene. This allows developers to include arbitrary 3D content in FXDiagram's 2D canvas. By

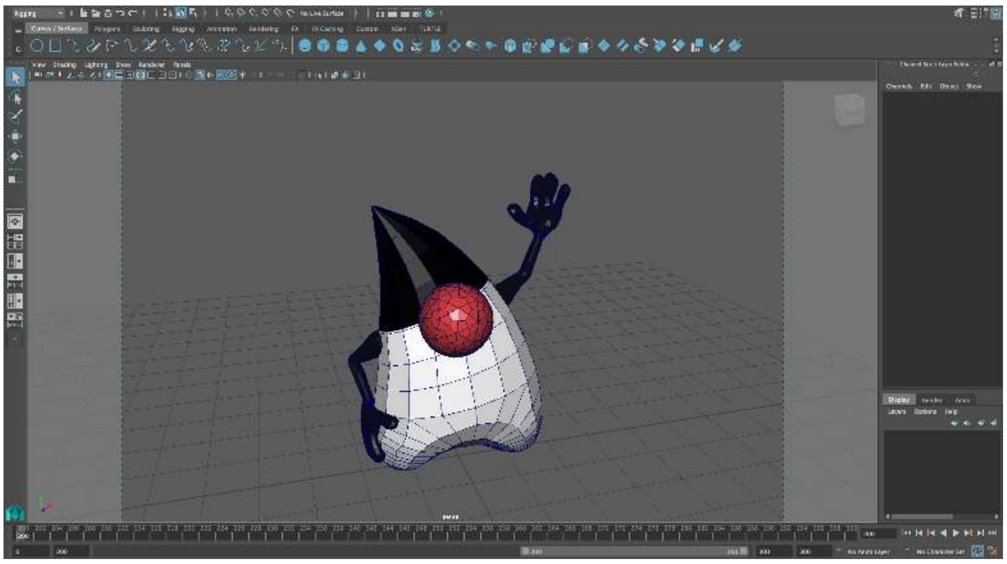


Figure 7.1: 3D Model in Maya

deactivating FXDiagram's tools used for handling selections, gestures, and mouse events, it is possible to implement custom event handlers for the 3D scene. This enables implementing the navigation behavior users expect when exploring 3D visualizations. Using `SubScene`, of course, has the major disadvantage that non of FXDiagram's features can be used anymore. This also includes the high-level mapping API and thus, the predefined integration between textual editor and graphical view. The integration needs to be added manually, similar as shown in the previous section.

When moving from 2D to 3D scenes, this also has an effect on the included media used. Images in 2D may need to be replaced by 3D models. Creating these more complex models may be tedious when using the Java API for creating 3D content. 3D models are hardly created using textual programming languages. 3D modeling is the process of creating a virtual three-dimensional model of some physical object by using specialized software. Well-known applications for 3D modeling, animation, and rendering are, for example, Blender [9] and Autodesk Maya [10]. Maya is mostly used for creating 3D assets for use in cinematic films, television, and game development. 3D models are often based on polygons, which consist of geometry based on vertices, edges, and faces. By combining many individual polygons, polygon meshes are created, which can later be rendered. A 3D model in Maya is shown in Figure 7.1. The shown wireframe lines represent the edges of each individual polygon.

Ideally, it should be possible to import 3D models, which are created in these modeling applications, into JavaFX. There are various model importers by InteractiveMesh [11] which allow the import of models in 3ds, COLLADA, FXML8, OBJ, STL, and X3D formats. The STL and

[9]`blender.org`

[10]`autodesk.com/products/maya`

[11]`http://www.interactivemesh.org/models/jfx3dbrowser.html`

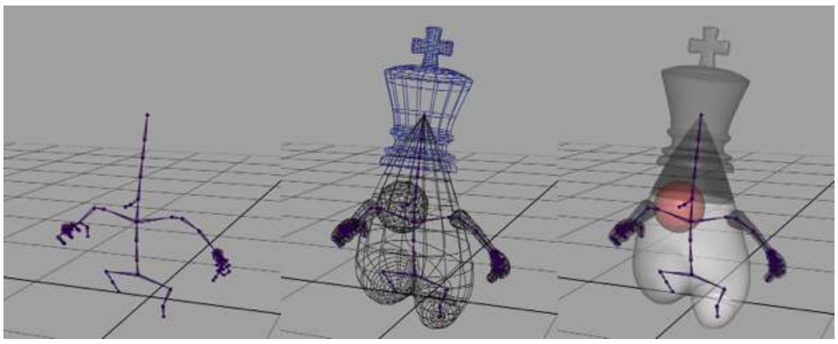


Figure 7.2: 3D Model including the skeleton

OBJ importers were used in Use Case 1 in order to show complex 3D models of monsters, entries, and exits.

As discussed previously, dynamic visualizations using animations are a great way of visualizing behavior. Animation classes provided by JavaFX such as `TranslateTransition` or `RotateTransition` can easily be used also in 3D scenes. The walk cycle of the player during a level traversal simulation was implemented by means of a very simple sprite animation in the 2D visualization. A walk cycle in a 3D traversal visualization, however, is only possible if the 3D model is animated. Animation of 3D models, however, is a bit more complex again. 3D models are usually animated by once more using applications such as Maya.

3D models are animated by creating a transform hierarchy to which the 3D geometry is attached. This task is usually called rigging or character setup. Just like humans, 3D models need a skeleton for moving. The skeleton has to be attached to the initially created geometry. This process is called skinning. Figure 7.2 shows the rigged 3D model. The image was created by José Pereda[12].

Finally, the 3D model can be animated. Motion is defined by specifying the model's key poses and using interpolators for creating the frames between those key poses. A very simple walk cycle can already be created by only defining four key poses.

Once again, a way to import these animated 3D models would be worth striving for. In 2009, an experimental demo application called 3DViewer was created by Oracle for importing 3D models into JavaFX. 3DViewer supports Maya and OBJ files and is also capable of importing animated models. Basically, this importer translates the Maya dependency graph into a corresponding JavaFX dependency graph. Concerning animation, Maya's keyframes are converted to JavaFX timelines. Unfortunately we were not able to successfully import the animated 3D models created for the first use case. The importer only supports a subset of Maya's nodes and does not appear to be working correctly with models created with more recent versions of Maya. After all, 3DViewer is an experimental project for demonstration purposes and it is not actively maintained.

[12] `jperedadnr.blogspot.co.at`

To conclude, switching from 2D to 3D views introduces a lot of technical challenges which have to be addressed in order to provide developers with a reliable and usable framework for creating 3D visualizations.

CHAPTER 8

Evaluation

Use Case 1 was created to investigate on the suitability of using FXDiagram and JavaFX as technologies for implementing visualizations. Use Case 2 aims at exploring the usefulness of visualizations in practice and is utilized for evaluation purposes. This chapter introduces Use Case 2, and describes the evaluation method as well as the results of the evaluation.

8.1 Use Case 2

While we invented Use Case 1 for demonstrating the capabilities of FXDiagram, Use Case 2 intends to show the actual impact of visualizations in practice. We, therefore, needed a textual DSL that is used on a regular basis by engineers. An Austrian automotive supplier permitted us to work with a DSL that is concerned with integration tests of devices utilizing work benches. The work benches are used to analyze the measured devices properties in different states. It is often not obvious which states are reachable, which are not reachable, which states have already been covered, and which states are impasses. This represents an ideal use case for applying visualizations, and analyzing the impact of visualizations on users' comprehension.

The DSL and Requirements

The language is a state-machine-like language that uses multidimensional states. This means that a user can define multiple state variables and each state variable can take on a state of a given range. Events are represented in the form of one or more inputs, having a predefined range as well. A model's behavior is defined by given-when-then clauses which can also be nested. An example model is depicted in Listing 8.1. When taking a look at the example model, the language has a very concise syntax which lets developers quickly and precisely define the behavior in specific situations. There are, however, many aspects that might make it hard for a user to fully understand the model.

```
device example {
  public statevar DeviceState {Pause,Standby,Measure} = Pause;
  public statevar UserLevel {Manual,Remote} = Remote;
  input UserAction{SPAU,STBY,SMES,SMAN,SREM};

  given UserLevel = Manual when UserAction = SREM then UserLevel -> Remote;

  given UserLevel = Remote {
    given DeviceState != Measure when UserAction = SMAN then UserLevel -> Manual;

    given DeviceState = Pause {
      when UserAction = STBY then DeviceState -> Standby;
      when UserAction = SMES then DeviceState -> Measure;
    }

    given DeviceState = Standby {
      when UserAction = SPAU then DeviceState -> Pause;
      when UserAction = SMES then DeviceState -> Measure;
    }

    given DeviceState = Measure {
      when UserAction = SPAU then DeviceState -> Pause;
      when UserAction = STBY then DeviceState -> Standby;
    }
  }
}
```

Listing 8.1: Example model written in the DSL of Use Case 2

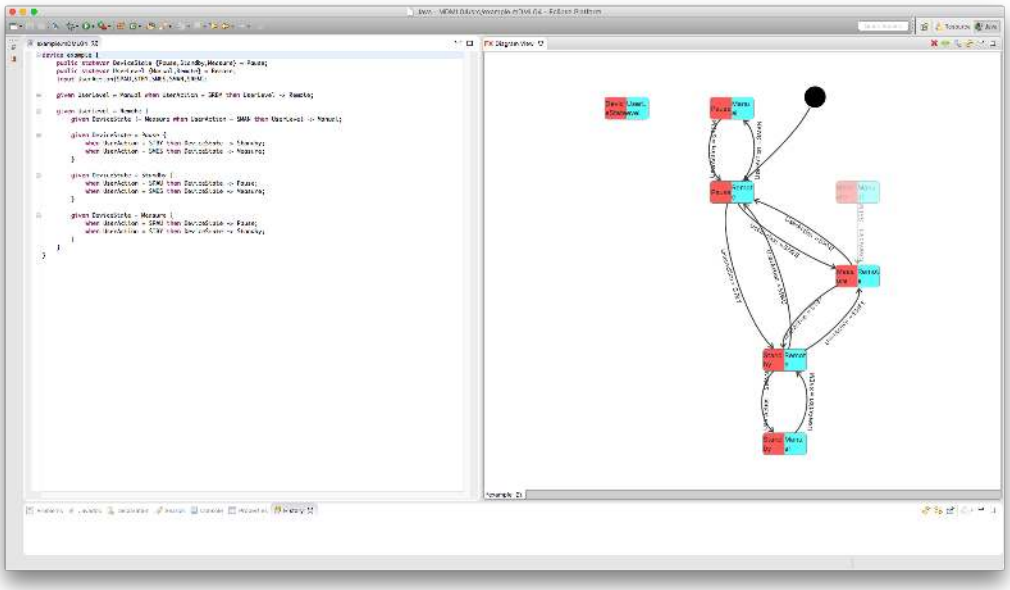
Firstly, the user is mainly interested in the overall system state. One system state is composed of all current states of the state variables. The rules in the model change only one state variable in one then-clause. It might be difficult for users to relate these changes to the overall state of the system. Secondly, negations such as `DeviceState != Measure` in given statements make it harder for users to understand in which states rules can actually be applied. When trying to understand the rule with the given negation, users first have to check, which states `DeviceState` can be in. Thirdly, when-statements can not only be an input event such as `UserAction = SREM`, but also state transitions, such as `UserLevel -> Remote`. This means that it is possible that a transition based on one rule, can trigger a transition defined in a completely different rule. Fourthly, nesting of rules might also result in models that are not very easily comprehensible.

Considering all of these aspects in one model, might make it really hard for developers to understand how a model works, to comprehend the consequences of changing, adding, or removing rules, and to spot impasses or errors. Exactly in situations like this visualizations should help developers. Visualizations, therefore, should make implicit transitions explicit, show reachable and unreachable states, highlight impasses, and support developers with tasks such as design, development, understanding code of other developers, and anomaly detection.

Implementation

We implemented a total of seven visualizations for the DSL described previously. In the following we shortly describe these visualizations and show the most important ones that were also used in the interviews.

One of the most important visualizations shows the textual model in the form of a state machine diagram. Nodes represent an overall state in the system and contain the values of all state variables. A node is, according to the number of state variables, divided into a respective



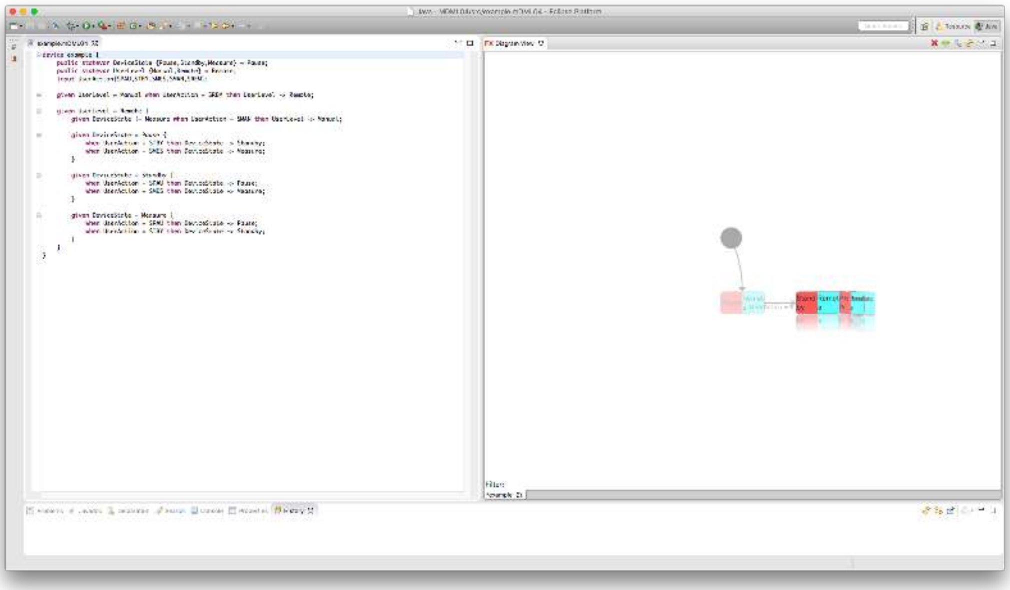
```
device example {
    public statevar DeviceState {Pause,StandBy,Measure} = Pause;
    public statevar UserLevel {Manual,Remote} = Remote;
    input UserAction{SRAU,STBY,SMES,SMAN,SREM};

    given UserLevel = Manual when UserAction = SREM then UserLevel -> Remote;

    given UserLevel = Remote {
        given DeviceState != Measure when UserAction = SMAN then UserLevel -> Manual;

        given DeviceState = Pause {
            when UserAction = STBY then DeviceState -> Standby;
            when UserAction = SMES then DeviceState -> Measure;
        }

        given DeviceState = StandBy {
            when UserAction = SPAU then DeviceState -> Pause;
            when UserAction = SMES then DeviceState -> Measure;
        }

        given DeviceState = Measure {
            when UserAction = SPAU then DeviceState -> Pause;
            when UserAction = STBY then DeviceState -> Standby;
        }
    }
}
```

Figure 8.1: Overall state diagram

number of subareas, having different background colors and containing the value of the state variable. The state machine's transitions are implicitly defined in the text by defining rules. The visualization makes these transitions explicit in the form of connections between nodes. The initial state is indicated similar to UML notation by a small solid filled circle. Impasses are highlighted by adding a red border to a node that constitutes an impasse. Unreachable states have a higher opacity than reachable ones. The overall state diagram for the example model given above is shown in Figure 8.1.

This visualization should mainly help developers get an overview of the model and show all overall states the system can be in. Explicitly showing possible transitions should make it easier for users to understand which states are reachable from a given state. The textual view is integrated with the diagram, in a way that selecting a connection in the diagram selects the rule that yielded this transition in the textual editor. By right-clicking on a rule in the textual editor and selecting a context menu entry, all corresponding connections of that rule can be revealed in the diagram view. This integration could, for example, help users trying to understand the model of another developer. This visualization should also help users during design and development by showing the structure of the system and highlighting impasses and unreachable states.

Based on the overall state diagram we implemented another visualization which aims at showing the execution of the state machine. After activating the visualization using the graphical context menu a panel with buttons for all possible input events is shown. Using the panel, the user can simulate input events. The current state is indicated by applying a blink-animation to the respective node. Previously visited nodes and connections are colored to highlight the



```
device example {
    public statevar DeviceState {Pause,StandBy,Measure} = Pause;
    public statevar UserLevel {Manual,Remote} = Remote;
    input UserAction{SPAU,STBY,SMES,SMAN,SREM};

    given UserLevel = Manual when UserAction = SREM then UserLevel -> Remote;

    given UserLevel = Remote {
        given DeviceState != Measure when UserAction = SMAN then UserLevel -> Manual;

        given DeviceState = Pause {
            when UserAction = STBY then DeviceState -> Standby;
            when UserAction = SMES then DeviceState -> Measure;
        }

        given DeviceState = StandBy {
            when UserAction = SPAU then DeviceState -> Pause;
            when UserAction = SMES then DeviceState -> Measure;
        }

        given DeviceState = Measure {
            when UserAction = SPAU then DeviceState -> Pause;
            when UserAction = STBY then DeviceState -> Standby;
        }
    }
}
```

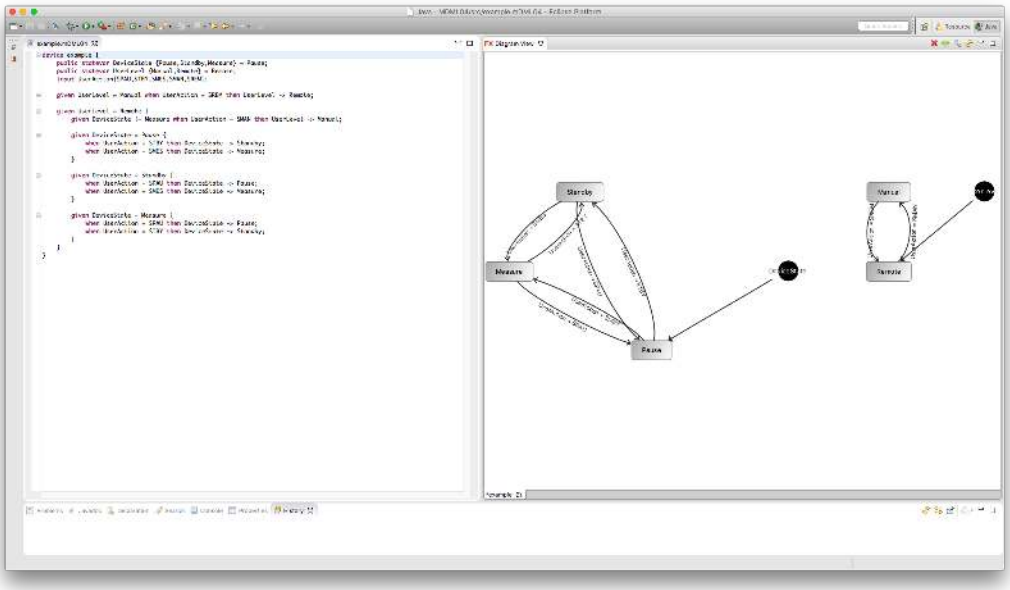
Figure 8.2: Diagram discovery

chosen paths. A chart visualization with statistical information about the theoretical number of nodes and actually reachable nodes can also be accessed through the graphical context menu.

The overall state diagram can, of course, get unclear very quickly if the number of state variables and states in the textual model increases. Our customer ensured that not too many state variables and states are used in their models, however, the case of having a large number of states has to be addressed. We used a feature provided by FXDiagram for handling this situation. Using *diagram discovery* users can explore a diagram and only view the parts they are interested in. Starting from the initial node users can discover neighbors of nodes using a chooser. When selecting a node in the chooser, the node and the respective connections are added to the diagram. This should help users get a better overview of diagrams containing a large amount of states. The choosing of a new neighbor node is shown in Figure 8.2.

While the overall state diagram aims at giving the user an overview of the whole system, we also implemented a visualization that shows only a smaller part of the system. The separate state visualization shows separate state machines for each state variable. This enables developers to understand which states of a state variable are actually reachable, and how these states can be reached. Figure 8.3 shows the visualization for the textual example model given above.

Often users may not be interested in the overall diagram, and also do not want to incrementally explore the overall diagram, by adding more and more states to an initial state. Users may only be interested in a state, and its neighbors. When selecting one of the neighbors the diagram



```
device example {
    public statevar DeviceState {Pause,StandBy,Measure} = Pause;
    public statevar UserLevel {Manual,Remote} = Remote;
    input UserAction{SPAU,STBY,SMES,SMAN,SREM};

    given UserLevel = Manual when UserAction = SREM then UserLevel -> Remote;

    given UserLevel = Remote {
        given DeviceState != Measure when UserAction = SMAN then UserLevel -> Manual;

        given DeviceState = Pause {
            when UserAction = STBY then DeviceState -> Standby;
            when UserAction = SMES then DeviceState -> Measure;
        }

        given DeviceState = StandBy {
            when UserAction = SPAU then DeviceState -> Pause;
            when UserAction = SMES then DeviceState -> Measure;
        }

        given DeviceState = Measure {
            when UserAction = SPAU then DeviceState -> Pause;
            when UserAction = STBY then DeviceState -> Standby;
        }
    }
}
```

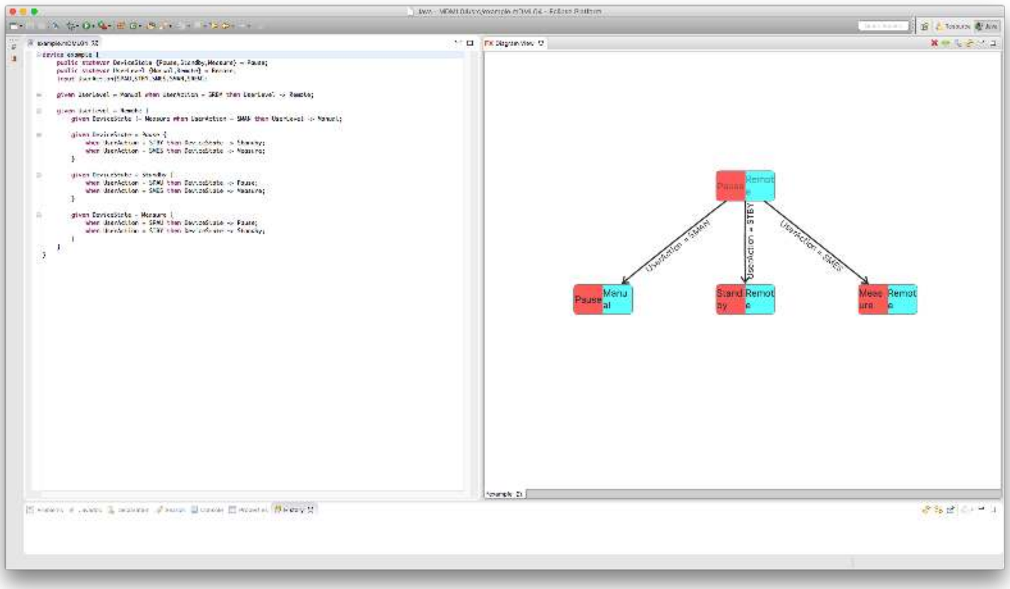
Figure 8.3: Separate state diagram

should be cleared and only the selected state and its neighbors should be shown. This is yet another way of exploring a diagram by displaying even a smaller part of the whole diagram. This should also help users in the process of trying to understand a model of another developer. Figure 8.4 shows this neighbor discovery visualization.

Finally, we wanted to give developers a possibility to analyze models. Two questions that may be of importance for developers are ‘Is the following state reachable?’, and ‘Is an impasse reachable from the following state?’. We implemented a visualization in which users first can interactively select the state to analyze and if a reachability and/or impass analysis should be performed. The user then gets the result of this analysis in the form of another diagram. One result of an analysis is shown in Figure 8.5. The blue connections highlight the path from the initial node to the selected node, whereas the red connections show the path to an impasse from the selected node. This visualization could be helpful during anomaly detection and debugging activities.

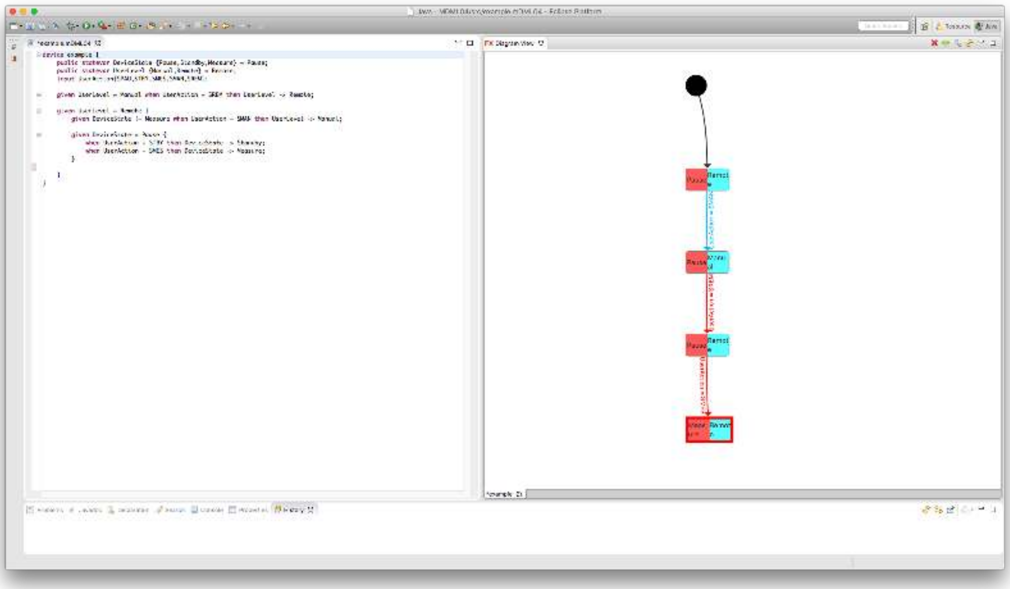
8.2 Interviews

Interviews are one of the best evaluation approaches for getting a detailed assessment by users for answering the research questions defined in the introduction. This section describes the interview method and design as well as the results.



```
device example {
    public statevar DeviceState {Pause,StandBy,Measure} = Pause;
    public statevar UserLevel {Manual,Remote} = Remote;
    input UserAction{SPAU,STBY,SMES,SMAN,SREM};

    given UserLevel = Manual when UserAction = SREM then UserLevel -> Remote;

    given UserLevel = Remote {
        given DeviceState != Measure when UserAction = SMAN then UserLevel -> Manual;

        given DeviceState = Pause {
            when UserAction = STBY then DeviceState -> StandBy;
            when UserAction = SMES then DeviceState -> Measure;
        }

        given DeviceState = StandBy {
            when UserAction = SPAU then DeviceState -> Pause;
            when UserAction = SMES then DeviceState -> Measure;
        }

        given DeviceState = Measure {
            when UserAction = SPAU then DeviceState -> Pause;
            when UserAction = STBY then DeviceState -> Standby;
        }
    }
}
```

Figure 8.4: Neighbor discovery



```
device example {
    public statevar DeviceState {Pause,StandBy,Measure} = Pause;
    public statevar UserLevel {Manual,Remote} = Remote;
    input UserAction{SPAU,STBY,SMES,SMAN,SREM};

    given UserLevel = Manual when UserAction = SREM then UserLevel -> Remote;

    given UserLevel = Remote {
        given DeviceState != Measure when UserAction = SMAN then UserLevel -> Manual;

        given DeviceState = Pause {
            when UserAction = STBY then DeviceState -> Standby;
            when UserAction = SMES then DeviceState -> Measure;
        }

    }
}
```

Figure 8.5: State analysis

Method and Design

The interviews were designed according to Boyce and Neale's [5] guide for conducting in-depth interviews. An in-depth interview is "a qualitative research technique that involves conducting intensive individual interviews with a small number of respondents to explore their perspectives on a particular idea, program, or situation". In-depth interviews are especially appropriate when detailed information about a person's thoughts are required, or when a new issue should be explored in detail. Investigating on if and how visualizations can improve the comprehension of a textual model requires detailed information about the thoughts of developers. We, therefore, think that conducting in-depth interviews is a suitable evaluation method for this thesis.

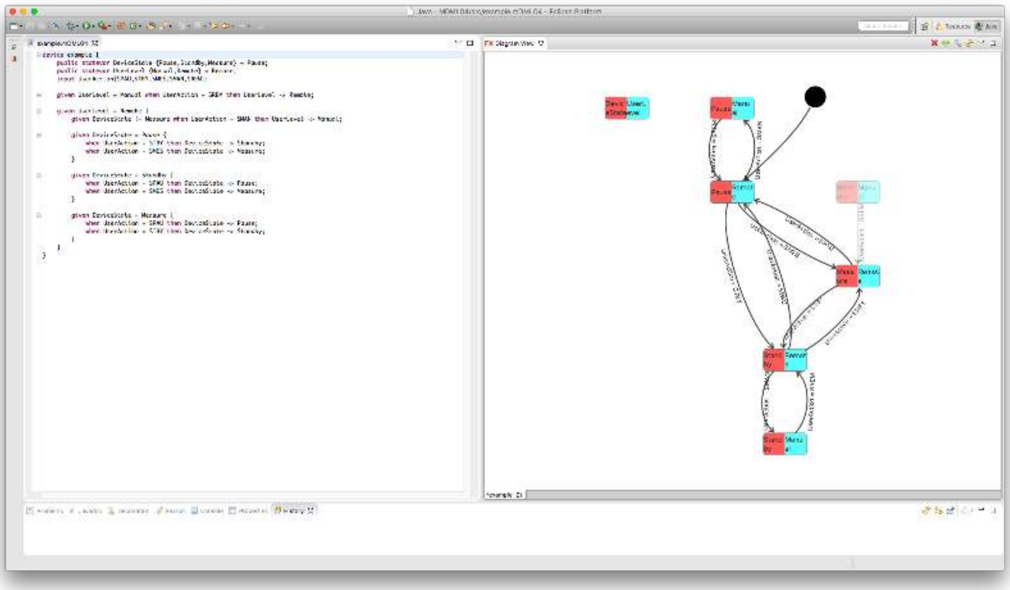
During the interview, four selected visualizations are shown to the interviewee. The selected visualizations are the overall state diagram, separate state diagram, neighbor discovery, and state analysis. After introducing the interviewee to a visualization, the interviewer always asks the same four questions. The first question addresses in how far the visualizations are useful and during which tasks the visualization can help developers. It should also show if the interviewees identify the same goals as the developer of the visualization. The second question explicitly asks if the visualization helps in understanding a textual model faster. The third question addresses RQ2 and asks the interviewee if the comprehension improvement of reports and visualizations differ. A potential report containing as much information as the visualization is shown to the interviewee on a sheet of paper. The fourth visualization-specific question asks the interviewee for remarks on the shown visualization. These four questions are asked after showing each of the four visualizations.

After that, three general questions are asked. The first general question aims at getting feedback on the navigation and interaction in the graphical view as well as the integration between textual and graphical view. The second general question asks for potential additional visualizations. The third general question should help with identifying problems the user has or had when working with the DSL, that are not solved by the visualizations shown previously. The user does not have to provide a solution in the form of a visualization to these problems. We refer the interested reader to the Appendix A for the full interview guide including the detailed questions and the reports.

In total, 6 in-depth interviews were conducted by the same interviewer over a period of two weeks. Each session was audio-recorded and the interviewer additionally wrote down the most important information on questionnaires. All interviewees are experts in using the respective textual DSL described above. The interviewees have, however, not been using visualizations for this DSL prior to the interview sessions and were confronted with our developed visualizations for the first time during the sessions.

Results

In the following, we present the results of the interview sessions, which, according to Boyce and Neale [5], should be done by using qualitative descriptors rather than trying to quantify the information. They state that "numbers and percentages sometimes convey the impression that results can be projected to a population, and this is not within the capabilities of this qualitative



```
device example {
    public statevar DeviceState {Pause,StandBy,Measure} = Pause;
    public statevar UserLevel {Manual,Remote} = Remote;
    input UserAction{SPAU,STBY,SMES,SMAN,SREM};

    given UserLevel = Manual when UserAction = SREM then UserLevel -> Remote;

    given UserLevel = Remote {
        given DeviceState != Measure when UserAction = SMAN then UserLevel -> Manual;

        given DeviceState = Pause {
            when UserAction = STBY then DeviceState -> Standby;
            when UserAction = SMES then DeviceState -> Measure;
        }

        given DeviceState = StandBy {
            when UserAction = SPAU then DeviceState -> Pause;
            when UserAction = SMES then DeviceState -> Measure;
        }

        given DeviceState = Measure {
            when UserAction = SPAU then DeviceState -> Pause;
            when UserAction = STBY then DeviceState -> Standby;
        }
    }
}
```

Figure 8.6: Overall state diagram (repeated from page 67)

research procedure" [5]. As the interview addressed four visualizations and some general questions we will also divide this section accordingly. The results presented in this section will be interpreted further in Chapter 10. We will also answer our research questions there.

Overall State Diagram

Most participants agreed that the Overall State Diagram (see Figure 8.6) is helpful. They had different motivations for classifying the visualization as helpful including

- the improved general overview of a model
- the depiction of state changes and state transitions
- the faster identification of unreachable states
- the depiction of testable parts
- the visualization of concurrent states
- a simplification in checking the correctness of existing models
- support during development of firmware behavior
- support during debugging
- simplified exploration of existing state machines

Furthermore, most interviewees agreed that the visualization helps to understand textual models faster. Several participants justified this feeling by stating that the visualization provides a better overview of the model, especially if the model size increases, and that relationships between states are shown.

When designing the visualizations we intended to make implicit transitions explicit, improve the overview of models, and show impasses as well as unreachable states with this visualizations. We also expected that these improvements should support users during development and during comprehending foreign code. Based on the results of the interview, the participants identified the visualization's main goals.

Concerning the comparison of the visualization and the report, all users agreed that the visualization is preferred. The prevalent feeling was that the visualization gives a better overview and is easier to understand. Users stated that "images are usually clearer and faster to read" ("Bilder sind in der Regel übersichtlicher und schneller zu lesen") and that "the graphical [representation] is more intuitive and easier to comprehend cases of high complexity" (" die Graphisch[e] [Darstellung] ist intuitiver und bei großer Komplexität leichter zu verstehen"). One user also stated that there is no difference if the model is small enough.

There are also some important suggestions for improvement. First of all, one user would like to have a possibility for filtering the visualization based on state variables, i.e. collapsing all state variables except for one. Secondly, some users identified problems with the used layout algorithm, e.g. unnecessary intersections of edges and misplaced or overlapping labels. Thirdly, users expected line breaks within state names. Finally, there were some remarks with respect to adding more information to the legend and changing colors for improving the usability.

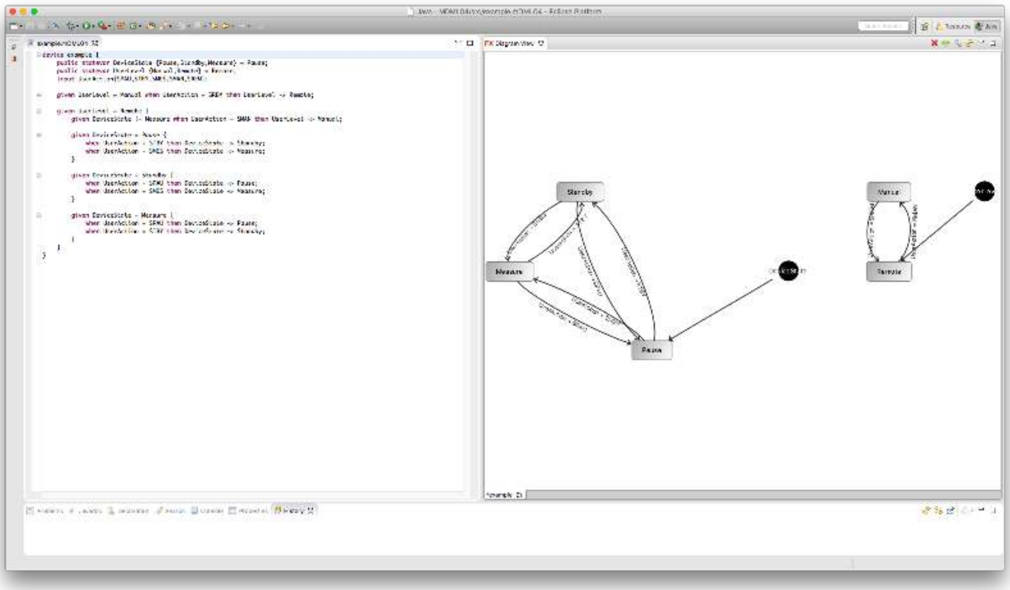
Separate State Diagram

All users agreed that the Separate State Diagram (see Figure 8.7) can be helpful. The interviewees stated that the visualization could support users with

- getting an overview by separating the state variables
- analyzing the completeness of models
- understanding models better because of reducing the complexity
- simplifying development in teams

Several participants felt that the visualization can help understand a textual model faster due to the fact that it can decrease the complexity of very large models. It may also be easier and faster to understand the behavior of a single state variable and might be useful for new developers trying to understand an existing model.

We expected that the visualization should reduce the complexity by only showing some parts of the overall model, should show reachable and unreachable nodes, and should support developers during the design and development phase. We can, therefore, say that the users identified the most important goals we had in mind while designing the visualization.



```
device example {
    public statevar DeviceState {Pause,StandBy,Measure} = Pause;
    public statevar UserLevel {Manual,Remote} = Remote;
    input UserAction{SFMU,STBY,SMES,SMAN,SREM};

    given UserLevel = Manual when UserAction = SREM then UserLevel -> Remote;

    given UserLevel = Remote {
        given DeviceState != Measure when UserAction = SMAN then UserLevel -> Manual;

        given DeviceState = Pause {
            when UserAction = STBY then DeviceState -> Standby;
            when UserAction = SMES then DeviceState -> Measure;
        }

        given DeviceState = StandBy {
            when UserAction = SFMU then DeviceState -> Pause;
            when UserAction = SMES then DeviceState -> Measure;
        }

        given DeviceState = Measure {
            when UserAction = SFMU then DeviceState -> Pause;
            when UserAction = STBY then DeviceState -> Standby;
        }
    }
}
```

Figure 8.7: Separate state diagram (repeated from page 69)

Concerning the preference for visualization or reports, several users preferred the visualization due to the clarity and improved overview. However, some users also favored the report as it does not have any disadvantages compared to the visualization.

All participants agreed that the visualization should include the dependencies between the state variables. Besides that, some users would prefer other colors (not only gray), and state variable names that are not cut off (which again is a space problem).

Neighbor Discovery

Several participants felt that the Neighbor Discovery visualization (see Figure 8.8) is helpful. They stated that the visualization could be potentially beneficial for

- getting to know a model
- highlighting the possible next states which is helpful in manual test case definition
- understanding larger models
- developing state machines

Some users stated that the visualization is not that useful because the previous states disappear when selecting a new node. Users, which require to see previous states and require a 'history' can use the State Discovery visualization, which has not been separately evaluated in the interviews. State Discovery was even mentioned by some users who 'accidentally' discovered

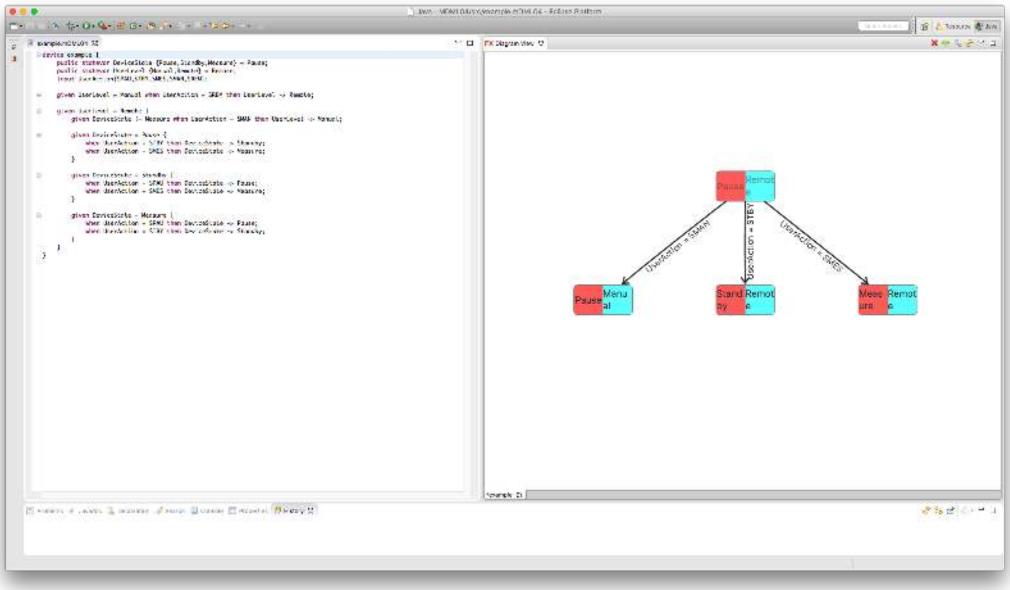

```
device example {
    public statevar DeviceState {Pause,StandBy,Measure} = Pause;
    public statevar UserLevel {Manual,Remote} = Remote;
    input UserAction{SFMU,STBY,SMES,SMAN,SREM};

    given UserLevel = Manual when UserAction = SREM then UserLevel -> Remote;

    given UserLevel = Remote {
        given DeviceState != Measure when UserAction = SMAN then UserLevel -> Manual;

        given DeviceState = Pause {
            when UserAction = STBY then DeviceState -> Standby;
            when UserAction = SMES then DeviceState -> Measure;
        }

        given DeviceState = StandBy {
            when UserAction = SFMU then DeviceState -> Pause;
            when UserAction = SMES then DeviceState -> Measure;
        }

        given DeviceState = Measure {
            when UserAction = SFMU then DeviceState -> Pause;
            when UserAction = STBY then DeviceState -> Standby;
        }
    }
}
```

Figure 8.8: Neighbor discovery (repeated from page 70)

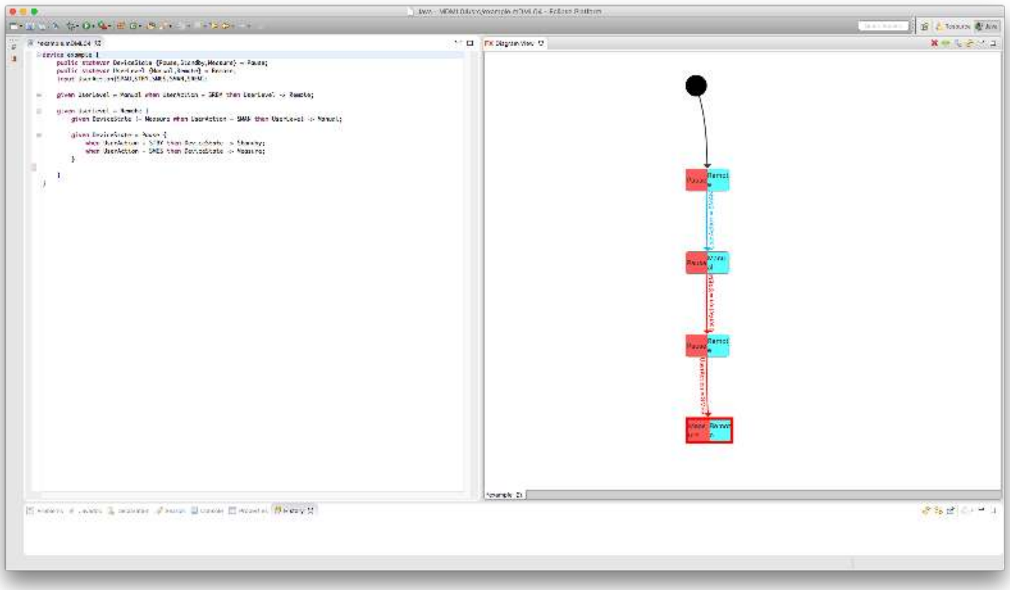
this visualization during the interview. We, therefore, conclude that exploration visualizations such as Neighbor Discovery and State Discovery may be helpful for developers.

One half of the users stated that the visualization shows a small part of the overall system in detail which could be helpful for understanding very large models faster. However, the other half did not find any advantages in using this visualization for understanding models faster.

We expected that the visualization should help users with exploring models, i.e. possible states and their neighbors, and understanding models faster. Some of the participants identified the exploration goal or agreed that the visualization supports faster comprehension of textual models. However, a similar amount of participants stated the exact opposite. Therefore, most participants did not identify our intended goals.

Concerning the comparison of the visualization and the report, all users agreed that the visualization is preferred. One user stated that he or she prefers the report for a small amount of neighbors and the visualization for states with many neighbors. There was a similar outcome with the Overall State Diagram. This leads to the assumption that visualizations may only be clearly superior to reports if the amount of information to visualize is large enough. We, however, think that the visualization is preferred here also because the interpretation is interactive, i.e. a state has to be selected in order to view its neighbors. This is simpler in a graphical representation by simply clicking on the desired node.

Several participants strongly suggested to include the predecessor states in the visualization. Furthermore, some participants suggested to combine this visualization with the Overall State Visualization by simply highlighting the currently selected node and hiding irrelevant nodes.



```
device example {
    public statevar DeviceState {Pause,StandBy,Measure} = Pause;
    public statevar UserLevel {Manual,Remote} = Remote;
    input UserAction{SPAU,STBY,SMES,SMAN,SREM};

    given UserLevel = Manual when UserAction = SREM then UserLevel -> Remote;

    given UserLevel = Remote {
        given DeviceState != Measure when UserAction = SMAN then UserLevel -> Manual;

        given DeviceState = Pause {
            when UserAction = STBY then DeviceState -> StandBy;
            when UserAction = SMES then DeviceState -> Measure;
        }

    }
}
```

Figure 8.9: State analysis (repeated from page 70)

State Analysis

The prevalent feeling was that the State Analysis visualization (see 8.9) could also be helpful. However, many participants noted that they prefer the Overall State Diagram as it provides a greater benefit (especially overview) to them. The interviewees stated that the visualization could support users

- during exploring a model or only parts of a model (in the case of larger models)
- with interactively defining and generating test cases
- with debugging and analysis of test results
- during path determination tasks
- with building the overall diagram step-by-step
- with becoming acquainted with the model through exploration
- with the comprehensibility of paths.

Some participants stated that exploring the model using this visualization could also lead to faster comprehension. Most participants, however, thought that the visualization does not give an overview of the model and, therefore, does not help understand the textual model faster. We also think that the participants often refer to the Overall State Diagram as it is also possible to analyze impasses and reachability requirements there.

Our goal with this visualization was to show if a specific state or an impasse is reachable. This should enable analyzing models and should help developers during debugging. As we did not primarily intend to speed up the comprehension process with this visualization we can say that the users identified the main purposes we thought of.

Concerning the preference for visualization or reports, the results are inconclusive. Some users prefer the graphical and some the textual representation. One user stated that he prefers the graphical one "although the textual representation is clear and comprehensible as well" ("aber die textuelle Darstellung ist auch übersichtlich und nachvollziehbar"). Given the fact that the reports of the model were short in this example but much longer for the Overall State Diagram, this again indicates that reports may be acceptable if they are not too long. One user also indicated that he or she prefers textual representations for state sequences. The fact that this visualization/report is also interactive challenges our assumption that visualizations are preferred if interaction is required.

Participants again pointed out the layouting issues discussed previously. Furthermore, a user stated that the text which is selected when clicking on a graphical node does not always match the user's expectation. More specifically, the inner most rule should be selected if a transition corresponding to a nested rule is selected.

General Results

The participants had many suggestions for improving the interaction between textual and graphical views. As previously mentioned, some interviewees pointed out that the inner most rule should be highlighted in the editor when selecting a transition. One user suggested to also show reports in addition to the visualization and editor. Some participants would like to have the possibility of applying filters in order to show only specific parts of a visualization, e.g. only some state variables. A suggestion that was given multiple times was the integration of two or more visualizations. The Neighbor Discovery could, for example, be integrated with the Overall State Diagram by highlighting the relevant parts and decreasing the opacity of other elements. Finally, users suggested implementing a UML-like sequence diagram and an additional tabular visualization.

CHAPTER 9

Related Work

Improving the comprehensibility of models by means of visualizations is an interdisciplinary task. Therefore, there is a large body of relevant work from related areas such as software visualization, program comprehension, model execution, and model simulation. However, few publications propose new visualization techniques for improving program comprehension and evaluate the newly introduced approach. At the time of writing we could not find any literature on the exact topic of improving comprehension of DSLs by utilizing visualization techniques. In the following we, therefore, introduce some of the relevant related work in the area of improving comprehension of software in general by using visualization.

9.1 Improving Comprehension by using Visualization

In 2000, Storey et al. [28] explored whether program understanding tools enhance or change the way developers understand programs. They think that tools should support users with their preferred strategy for understanding programs, rather than imposing a fixed strategy that may not be suitable. A user study was conducted that compared three tools for browsing source code and exploring software structures. In the study 30 participants used the tools for solving several program understanding tasks that require a broad range of comprehension strategies. The study was divided into three parts. In the first one the participants had to perform formal tasks on a given program. In the second one the participants had to complete a questionnaire, and in the third one an interview was conducted. The results revealed that in general the tools did enhance the users' preferred comprehension strategies while solving tasks. For example, the ability to view dependency relationships and switching between high-level views and source code were considered useful features. In some instances, however, the tools hindered the user's progress due to the fact that the tools did not support some comprehension strategies. This forced users to change their chosen comprehension strategy, which is undesirable. The interviews revealed that usability is also very important when using tools for improving program comprehension.

In 2007, Lange et al. [17] claimed that existing tooling does not offer sufficient support for understanding UML models and evaluating their quality. They implemented four additional interactive views and conducted an experiment, validating whether there is a difference between existing views and the new views with respect to comprehension correctness and comprehension effort. The MetaView view was used for visualizing relations between diagrams with different levels of abstraction. ContextView shows a model element, and all the model elements it relates to in all existing diagrams. In MetricView three different metrics are visualized on top of a regular class diagram. Finally, the UML-CityView combines MetaView with MetricView, and shows a colored box indicating the value of the metric on top of the model elements. 100 participants completed a questionnaire about comprehension tasks on a given model. The results showed that the effort needed to complete tasks is reduced by 20% and the correctness of the comprehension tasks is increased by 4.5% .

Cornelissen et al. [8] conducted a controlled experiment, quantitatively evaluating the added value of execution trace visualization for program comprehension in 2011. The large amount of trace data usually created by applications can only be used for improving comprehension by applying automatic reductions and visualization techniques. Execution traces are most often visualized in the form of directed graphs or UML sequence diagrams. The 34 participants, which were divided into two groups had to complete eight typical tasks in the context of software maintenance which aimed at gaining an understanding of a representative system. The control group completed the tasks only by using the Eclipse IDE, while the experimental group used the Eclipse IDE as well as EXTRAVIS, a tool for the visualization of large traces. The results of the experiment show a 22% decrease in time needed for a given task and a 43% increase in correctness of the results for the group using execution trace visualization.

In 2011, Wettel et al. [32] analyze the capabilities of using the city metaphor, a 3D software visualization approach, for improving the comprehension of programs. Using CodeCity, a tool for creating visualizations, they visualize software systems as three-dimensional cities, where classes are buildings and packages are districts. They map the number of methods on the height of buildings, the number of attributes on the base size, and the number of lines of code on the color of buildings. This leads to visualizations such as shown in Figure 9.1. 41 participants were involved in the experiment and had to complete tasks in adaptive and perfective maintenance. Each participant was assigned to either the control group or the experimental group. The control group worked with Eclipse IDE and Excel spreadsheets containing metrics and design problem data. The experimental group worked with CodeCity. The results of the experiment show that the visualization approach leads to an improvement in correctness (+24%) and completion time (-12%) over the Eclipse IDE and spreadsheet approach. Using CodeCity turned out to be very valuable when working on tasks which require getting an overview of the system. However, the group using CodeCity did not perform better on tasks which required very precise answers.

In 2013 Sharafi et al. [25] empirically investigated the efficiency of graphical representations against textual ones in presenting software requirements. They conducted an experiment with 28 participants and used an eye tracking system for analyzing the exact location and duration

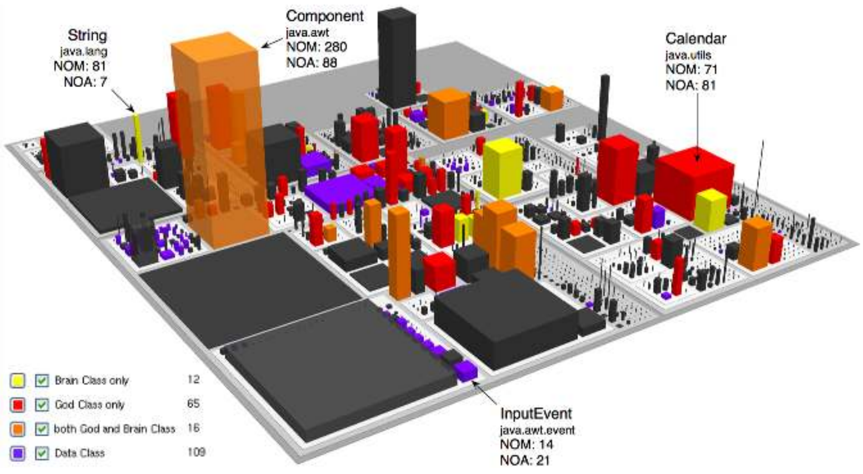


Figure 9.1: Software system visualized in CodeCity (taken from [32])

of where the subject is looking. Using the gathered data it is possible to compute the subject's visual effort and visual paths while reading requirements. The experiment compared structured textual representations of requirements with graphical representations created by a tool called TROPOS. By analyzing the participants' answers on comprehension questions and the data gathered by using the eye tracking system the researchers calculated answer accuracy, overall time, and the visual effort. Subjects were randomly assigned to four groups. Each subject worked on one model using the textual representation, on another model using the graphical representation, and on a third one using both textual and graphical representations. The results show no statistically-significant differences between the representation types when considering accuracy. However, subjects spent more time and effort while working with the graphical representation. The participants did state that they preferred working with the graphical view, however, performed the tasks more efficiently while working with the textual representation. The experiment also showed that subjects perform significantly better with respect to time and effort spent while working with the mixed view after first working with the two views separately. This shows that working with graphical representations requires some training before they can be used beneficially.

In 2013 Sorva et al. [27] provided a comprehensive survey of program visualization systems intended for teaching beginners about the runtime behavior of computer programs. The review showed that program visualization systems for beginners are very often short-lived research prototypes, which are discarded as soon as the system had been constructed or an evaluation study was carried out. The survey also revealed that many of the visualization systems have been evaluated informally. It is, therefore, unclear whether using these systems actually has resulted in significant learning gains. Usually the more rigorously performed experiments were conducted by the authors themselves, often in their own teaching. Many of these experiments' results were positive and showed that the visualizations served some purpose. All in all, Sorva et al. conclude that the existing literature on program visualization systems largely supports the use of program visualization in introductory programming education, but that it is not possible to draw more nuanced conclusions with respect to learner engagement.

9.2 Other Related Work

In 2009 Hussein et al. [14] explored the rather unusual but highly interesting approach of supplementing visualizations by sonification in order to improve code comprehension. They implemented an Eclipse plugin which generates sonifications when hovering over a method. Three values, representing the number of code lines, total number of method calls in this method, and the total number of calls to a `java.utils.io` method were used as an information source. The plugin then generated rain from the left speaker, water stream from the right speaker, and cello from the central speaker representing these three values. The sonic cue volume increases with the increased numeric value of the given information. In the experiment 10 participants answered questions about an unfamiliar code base first by using a visualization and then the sonification. The results indicate that information sonification can be as effective as information visualization at some levels, including correctness and comprehension. However, most of the participants prefer using visualization over sonification in practice. Sonification, therefore, is rather a supplement to than a replacement for visualization.

CHAPTER 10
Summary and Conclusion

This thesis aimed at investigating the potential of using visualizations for improving the comprehension of DSLs. We presented general information about DSLs as well as the technical language concepts of DSLs. We analyzed and described ways of visualizing the Semantic Model such as the Concrete Syntax, Visualizations, and Reports. The need for a base technology which could be used for creating high quality visualizations lead to an analysis of existing frameworks based on Draw2D as well as new frameworks based on JavaFX. Due to the fact that there was not any literature about creating visualizations using JavaFX this thesis is based on two use cases. The first use case aimed at exploring the capabilities of using FXDiagram and JavaFX for creating visualizations in the domain of game level design. Based on the experience gained from the first use case we presented a conceptional visualization architecture as well implementation details and implementation challenges. The second use case intended to explore the practical relevance of visualizations in the domain of automotive testing and served as an evaluation use case. We present and interpret our results in the following section and answer the research questions defined in Chapter 1. Finally, we compare this thesis to the discussed related work presented in Chapter 9 and present some remaining challenges which can be addressed in future work.

10.1 Interpretation of Results

We will interpret our results in the process of answering the research questions in detail below.

RQ1. Do visualizations increase users' comprehension of models?

The results presented in Chapter 8 indicate that visualizations can support users of DSLs in various ways. All of the participants agreed that the main visualization — the Overall State Diagram — provided a benefit in some form or another. The answers concerned with the other visualizations did not yield such clear result because some users assessed the visualizations as being helpful and others did not consider them as helpful. Nevertheless, we can say that also these visualizations provided benefits to some users. This indicates that the helpfulness of a visualization greatly depends on the user's way of working and even on the user's taste. Both are very subjective which means that, just like in traditional software engineering, it is very important to define the exact requirements and design of visualizations in close cooperation with future users. When designing the visualizations we had our ideas of what goals could be fulfilled and where the visualization could be used. The results of the interview showed that some users identified these goals, however, other users identified very different goals and application areas.

Therefore, the extent to which a visualization is helpful for a user greatly depends on the user and the quality of the visualization. Different users require different visualizations. It is, thus, reasonable to provide multiple visualizations for one model, similarly to offering multiple CSs like described in Section 3.1. That way users can choose the perfect visualization based on their needs. This can, of course, be work-intensive and expensive and has to be evaluated in advance. Considering the users detailed requirements and suggestions, however, is vital regardless of creating multiple or just one visualization.

When providing multiple visualizations it is important to find an intuitive way of integrating them. In Section 3.2 we mention that integrating the textual view and the visualization is important. However, the interviews revealed that it is also very important for users to conveniently switch between different visualizations and not get 'lost' in the process of doing so. For example, users did not immediately see the relation between the State Analysis and the Overall State Diagram. Integrating analysis functionality into the Overall State Diagram could help users not lose the context of their operations.

Based on the results of the interview we are convinced that visualizations can increase users' comprehension of models, especially when considering the factors described above.

RQ2. Are visualizations superior to reports which highlight a particular aspect in textual notation?

This question aimed at investigating in how far it is necessary to provide visualizations for the results of a model interpretation or if a textual report is sufficient as well. The interview results suggest that most users prefer visualizations to reports. There were, however, some participants stating that the textual report does not have any disadvantages compared to the visualization, especially in the case of small models. We, therefore, conclude that visualizations may be superior to reports, especially if the amount of information resulting from the interpretation is very large. This basically boils down to the general question of in how far visualizations can present data in a more comprehensible way than texts can. Furthermore, which one is preferred may also depend on in how far the interpretation is interactive or not. For example, the Neighbor Discovery visualization could also be replaced by an interactive command-line-like report. This report could output the neighbors in textual form and also take the next state as a textual input. This leads to the general discussion of command-line versus graphical interface. Therefore, once again it greatly depends on the users subjective preferences. We, however, believe that visualizations are superior to reports especially when a large amount of interpretation data has to be shown.

RQ3. Is JavaFX a feasible technology for implementing visualizations for DSLs?

The interviews' results revealed that usability and aesthetics are very important for users. There were many suggestions for changing colors, adding more information to the legend, improving the routing of edges, and enhancing positioning of nodes and labels. This strengthens our assumption that a base technology used for creating visualizations should be capable of addressing usability requirements. We presented the capabilities of JavaFX and FXDiagram in Chapter 4 and have to state that we were not restricted during the development of most of the visualizations. There were some problems when creating 3D content, however, we are, not sure if there are base technologies that make this significantly easier. FXDiagram provides many features out of the box including basic integration between textual and graphical view, and all major navigation and interaction features. We cannot claim that FXDiagram is clearly superior to any of the frameworks discussed in Chapter 5 or GEF4. However, we would recommend FXDiagram for the development of visualizations as it focuses on usability, provides much functionality out of the box, and has a suitable abstraction level. Furthermore, it does not restrict developers and allows to leverage the full potential of JavaFX.

10.2 Comparison with Related Work

As mentioned in Chapter 9 we could not find any literature on the exact topic of using visualizations in order to improve the comprehension of DSLs. Therefore, the main difference between the related work and the work at hand is that related work addresses visualization in software in general, while this thesis focuses on DSLs. Furthermore, most of the related work is mainly concerned with using tools for visualizing very general aspects, such as software structures and architecture, dependencies, or metrics. This thesis, however, deals with highly-customized domain-specific visualizations which do not only focus on visualizing structural aspects of the textual model, but rather highlighting aspects that can be useful for the developer in specific situations. Highlighting aspects often requires an interpretation of the underlying model often under consideration of the languages semantics and the domain. Most of the related work had more sophisticated experiments for evaluating the visualization approaches and could also quantitatively present results. The visualizations created for this thesis were evaluated by interviews and qualitative descriptors.

10.3 Limitations and Future Work

There are two main concerns about the validity of our results for RQ1. Firstly, we only constructed visualizations for one language in a specific domain. The results could theoretically be very different with a different language or a different domain. Future experiments should, therefore, analyze visualizations of multiple languages and domains. Secondly, the results are based solely on the interviewees' impressions on the visualization's usefulness. Future experiments should test the actual improvement by using questions giving evidence that developers working with visualizations can actually comprehend models better or faster.

Concerning RQ2 we established the assumption that visualizations are superior to reports especially if there is a large amount of information resulting from the interpretation. This assumption could be addressed in a dedicated experiment. As this thesis' focus was on the construction of visualizations, the main effort was put into the visualizations. The reports were only mockups and not highly sophisticated. A future experiment should, therefore, use well-constructed and well-designed reports in order to produce reliable results. It would also be interesting to explore limitations and advantages of reports and visualizations and show whether it is useful to combine the two.

RQ3 was answered mainly with our experiences gained during this thesis' practical part. This thesis was not intended to be a technology comparison. It would, however, be necessary to implement the same visualization with multiple languages and frameworks in order to find the best technology for implementing visualizations.

Further challenges include the analysis and implementation of various filters for visualizations such as suggested by the participants of the interviews as well as finding solutions to the technical challenges described in Section 7.3.

APPENDIX A

Interview documents

Interview Guide

Zweck des Interviews

Die Aufgaben eines Softwareentwicklers — unter anderem Design, Implementierung, Verständnis von Code anderer Entwickler, Fehlerfindung sowie Wartung — erfordern alle grundlegendes Verständnis des Programms. Das Ziel der Arbeit und dieses Interviews ist es, zu untersuchen, ob Visualisierungen von domänen-spezifischen Sprachen das Verständnis von textuellen Programmen verbessern können und Entwickler dadurch bei ihren Aufgaben unterstützt werden können. Zusätzlich soll herausgefunden werden, ob zur Verbesserung des Verständnisses *graphische* Repräsentationen („Visualisierungen") verwendet werden sollten oder ob eine Analyse / Interpretation des Programms mit *textueller* Ausgabe des Ergebnisses (ein sogenannter „Report") ausreichend ist.

Ablauf Einführung (vor Interview)

- Begrüßung
- Vorstellung Interviewer
- Zweck des Interviews beschreiben (siehe oben)
- Vertraulichkeit (siehe Einwilligungserklärung)
- Erklärung zum Ablauf des Interviews (siehe unten) und der Dauer (ca. 30-45 Minuten)
- Möglichkeit für Fragen
- Unterschrift bei Einwilligungserklärung

Ablauf Interview

- Interviewer öffnet Programm **interview.mDML04**
- Allgemeine Einführung in Entwicklungsumgebung durch Interviewer (Texteditor vs. graphische Ansicht, wie kann graphische Ansicht upgedated werden)
- Öffnen der ersten Visualisierung (**Overall State Diagram**)
- Dem Interviewee einige Minuten Zeit geben um sich mit der Visualisierung vertraut zu machen (Programmcode ändern lassen und darauf hinweisen, dass die graphische Ansicht immer upgedated werden kann; Programm evtl. auch so abändern lassen, dass Sackgassen entstehen)
- Anschließend die Fragen zur Visualisierung 1 (**1.1.-1.4.**) stellen und Antworten des Interviewees notieren
- Öffnen der zweiten Visualisierung (**Separate State Diagram**)
- *Vorgehen analog zur vorherigen Visualisierung.* **Fragen: 2.1.-2.4.**
- Öffnen der dritten Visualisierung (**Neighbor Discovery**)
- *Vorgehen analog zur vorherigen Visualisierung.* **Fragen: 3.1.-3.4.**
- Öffnen der vierten Visualisierung (**State Analysis**)
- *Vorgehen analog zur vorherigen Visualisierung.* **Fragen: 4.1.-4.4.**
- Abschließend allgemeine Fragen (**5-7**) stellen
- Noch Fragen oder Kommentare?
- Für Interview bedanken

Details zu den Fragen (nur für Interviewer)

Es werden zuerst zu vier ausgewählten Visualisierungen jeweils die gleichen vier Fragen gestellt. Abschließend gibt es noch drei allgemeine Fragen, welche sich nicht auf spezifische Visualisierungen beziehen.

Die Fragen 1.1., 2.1., 3.1. und 4.1. untersuchen, ob die Visualisierungen für den Interviewee sinnvoll sind, wie genau sie ihm helfen und in welchem Bereich die Visualisierung hilfreich sein kann. Die Visualisierungen haben immer ein Ziel und versuchen etwas Bestimmtes zu erreichen (siehe unten). Es wäre interessant, ob die Interviewees durch die Beantwortung dieser Fragen ähnliche Ziele identifizieren oder ob die Visualisierungen für sie andere (oder gar keine) Ziele erreichen. Die untenstehenden Ziele sollten dem Interviewee gegenüber deshalb natürlich nicht direkt erwähnt werden.

Ziele der Visualisierungen (*aus Sicht des Entwicklers der Visualisierungen*):
Visualisierung 1: Explizit-machen von impliziten Transitions, Übersicht über die erreichbaren Gesamtzustände verbessern, Sackgasse und Möglichkeiten in eine Sackgasse zu laufen aufzeigen, Unterstützung bei Design und Entwicklung sowie beim Verstehen fremden Codes
Visualisierung 2: Explizit-machen von impliziten Transitions, Komplexität durch Zeigen eines Teiles des Gesamtsystems verringern, Erreichbarkeit von States einzelner Variablen veranschaulichen und nichtbenötigte/nicht erreichbare States identifizieren, Unterstützung bei Design und Entwicklung
Visualisierung 3: Explorativ Überblick über Programm bzw. States und deren Nachbarn erhalten, Bedeutung des Programms schneller verstehen
Visualisierung 4: Schnelles Aufzeigen ob ein bestimmter Zustand erreichbar ist, wie dieser erreichbar ist und ob von dem gewählten Zustand eine Sackgasse erreichbar ist, genauere Analyse des Programms ermöglichen und Entwickler bei z.B. Fehlerfinden / Debugging unterstützen

Die Fragen 1.2., 2.2., 3.2. und 4.2. zielen konkret darauf ab, ob die Interviewees glauben, dass man durch die Visualisierungen textuelle Programme schneller oder besser verstehen kann.

Die Fragen 1.3., 2.3., 3.3. und 4.3. sollen feststellen ob Visualisierungen (*graphische* nicht-editierbare Repräsentationen) überhaupt notwendig sind, oder ob auch ein *textueller* Output einer Analyse/Interpretation ausreicht um das Verständnis zu verbessern. Zu diesen Fragen gibt es zum Programm **interview.mDML04** textuelle Darstellungen der Visualisierungen im Zusatzdokument **Reports.pdf**.

Fragen 1.4., 2.4., 3.4. und 4.4. sind für Anmerkungen zu den entsprechenden Visualisierungen gedacht.

Frage 5 dient dazu Feedback über die Usability bzgl. Navigation und Interaktion in der graphischen Ansicht, Integration zwischen textueller und graphischer Ansicht etc. einzuholen.

Frage 6 und 7 sollen potentielle Möglichkeiten für weitere Visualisierungen identifizieren. Frage 6 fragt dabei explizit nach möglichen Visualisierungen und Frage 7 nach Problemen bei der Verwendung der DSL (für die der Interviewee aber keine Lösung in Form einer Visualisierung haben muss).

Öffnen der Visualisierungen

Nach Starten der IDE können die Visualisierungen wie folgt geöffnet werden:

Visualisierung 1: Rechtsklick auf „device Interview“ im Texteditor -> „Show in FXDiagram as...“ -> „Overall State Diagram (Device)“
Visualisierung 2: Rechtsklick auf „device Interview“ im Texteditor -> „Show in FXDiagram as...“ -> „Separate State Diagram (Device)“
Visualisierung 3: Rechtsklick auf „device Interview“ im Texteditor -> „Show in FXDiagram as...“ -> „Overall State Neighbor Discovery (Device)“
Visualisierung 4: Zuerst Visualisierung 1 öffnen. Anschließend in der graphischen View rechtsklicken um das graphische Kontextmenü zu öffnen. Im Kontextmenü das „Kompass-Symbol“ (Reachability and Impasse Analysis (R)) auswählen. Die Visualisierung kann durch erneutes Klicken auf das „Kompass-Symbol“ wieder geschlossen werden.

Interaktion und Integration

Navigation sowie Positionierung der graphischen Elemente erfolgt durch Drag & Drop. Zoom sowie Undo und Redo können durch Standard-Shortcuts oder das graphische Kontextmenü durchgeführt werden.

Sofern möglich und sinnvoll sind die textuelle und die graphische Ansicht verbunden. Selektiert man z.B. eine Connection in der graphischen Ansicht, wird die entsprechende textuelle Regel im Editor markiert. Mit Rechtsklick auf eine Regel im Texteditor und „Reveal rule in Diagram“ werden in der graphischen Ansicht Connections markiert welche aufgrund der gewählten Regel zustande kommen.

Die graphische Ansicht kann durch den Benutzer nach Änderung des Programmcodes upgedated werden: Im rechten oberen Bereich der graphischen Ansicht befindet sich das Schraubenschlüssel-Symbol (🔧). Klickt man darauf wird die graphische Ansicht aktualisiert.

Anmerkung: Bitte achten Sie darauf, dass die Diagramme/graphische Ansicht **nicht** gespeichert werden. Das Speichern bzw. Laden von komplexeren Visualisierungen ist bei Verwendung von FXDiagram nicht einfach umzusetzen und wurde bei den Visualisierungen nicht implementiert. Falls sich eine Visualisierung nicht öffnen lässt (es wird nur eine leere weiße Ansicht geöffnet) liegt dies höchstwahrscheinlich daran, dass das Diagramm zuvor gespeichert wurde. Dies lässt sich leicht dadurch erkennen, dass im Package Explorer ein fxd-File vorhanden ist. Nach Löschen dieses Files sollte wieder alles fehlerfrei funktionieren.

Einwilligungserklärung zur Erhebung und Verarbeitung personenbezogener Interviewdaten

Diplomarbeit:	Visualizing Domain-Specific Languages Utilizing JavaFX
Diplomand:	Alexander Altenhuber, BSc
Universität:	TU Wien

Interviewer:	____
Datum:	____

Ich erkläre mich dazu bereit, im Rahmen der genannten Diplomarbeit an einem Interview teilzunehmen. Ich wurde über das Ziel und den Verlauf des Projekts informiert. Ich kann das Interview jederzeit abbrechen, weitere Interviews ablehnen und meine Einwilligung in eine Aufzeichnung und Niederschrift des Interviews zurückziehen, ohne dass für mich dadurch irgendwelche Nachteile entstehen.

Ich bin damit einverstanden, dass das Interview mit einem Aufnahmegerät aufgezeichnet und dann von den Mitarbeiterinnen und Mitarbeitern des Studienprojekts in Schriftform gebracht wird. Für die weitere wissenschaftliche Auswertung des Interviewtextes werden alle Angaben zu meiner Person aus dem Text entfernt und/oder anonymisiert. Mir wird außerdem versichert, dass das Interview in wissenschaftlichen Veröffentlichungen nur in Ausschnitten zitiert wird, um sicherzustellen, dass ich auch durch die in den Interviews erzählte Reihenfolge von Ereignissen nicht für Dritte erkennbar werde.

Vorname, Nachname:	____
Ort, Datum, Unterschrift:	____

Visualisierung 1: Overall State Diagram

1.1. Ist die Visualisierung für Sie hilfreich? Wenn ja: Weshalb ist die Visualisierung für Sie hilfreich und bei welchen Tätigkeiten kann Sie die Visualisierung unterstützen? Wenn nein: Weshalb ist die Visualisierung nicht hilfreich?

1.2. Inwiefern hilft die Visualisierung dabei, das entsprechende textuelle Programm schneller oder besser zu verstehen?

1.3. Visualisierung vs. Report (Report 1 im Zusatzdokument **Report.pdf**): Präferieren Sie die graphische oder textuelle Repräsentation? Bitte begründen Sie Ihre Antwort.

1.4. Haben Sie Anmerkungen oder Verbesserungsvorschläge zu dieser Visualisierung?

Visualisierung 2: Separate State Diagram

2.1. Ist die Visualisierung für Sie hilfreich? Wenn ja: Weshalb ist die Visualisierung für Sie hilfreich und bei welchen Tätigkeiten kann Sie die Visualisierung unterstützen? Wenn nein: Weshalb ist die Visualisierung nicht hilfreich?

2.2. Inwiefern hilft die Visualisierung dabei, das entsprechende textuelle Programm schneller oder besser zu verstehen?

2.3. Visualisierung vs. Report (Report 2 im Zusatzdokument **Report.pdf**): Präferieren Sie die graphische oder textuelle Repräsentation? Bitte begründen Sie Ihre Antwort.

2.4. Haben Sie Anmerkungen oder Verbesserungsvorschläge zu dieser Visualisierung?

Visualisierung 3: Neighbor Discovery

3.1. Ist die Visualisierung für Sie hilfreich? Wenn ja: Weshalb ist die Visualisierung für Sie hilfreich und bei welchen Tätigkeiten kann Sie die Visualisierung unterstützen? Wenn nein: Weshalb ist die Visualisierung nicht hilfreich?

3.2. Inwiefern hilft die Visualisierung dabei, das entsprechende textuelle Programm schneller oder besser zu verstehen?

3.3. Visualisierung vs. Report (Report 3 im Zusatzdokument **Report.pdf**): Präferieren Sie die graphische oder textuelle Repräsentation? Bitte begründen Sie Ihre Antwort.

3.4. Haben Sie Anmerkungen oder Verbesserungsvorschläge zu dieser Visualisierung?

Visualisierung 4: State Analysis

4.1. Ist die Visualisierung für Sie hilfreich? Wenn ja: Weshalb ist die Visualisierung für Sie hilfreich und bei welchen Tätigkeiten kann Sie die Visualisierung unterstützen? Wenn nein: Weshalb ist die Visualisierung nicht hilfreich?

4.2. Inwiefern hilft die Visualisierung dabei, das entsprechende textuelle Programm schneller oder besser zu verstehen?

4.3. Visualisierung vs. Report (Report 4 im Zusatzdokument **Report.pdf**): Präferieren Sie die graphische oder textuelle Repräsentation? Bitte begründen Sie Ihre Antwort.

4.4. Haben Sie Anmerkungen oder Verbesserungsvorschläge zu dieser Visualisierung?

Allgemeines

5. Was könnte man bei den gezeigten Visualisierungen in Bezug auf Navigation und Interaktion in der graphischen Ansicht sowie Integration zwischen textueller und graphischer Ansicht verbessern?

6. Welche zusätzlichen Visualisierungen wären für Sie noch hilfreich?

7. Gibt es Probleme bei der Verwendung der domänen-spezifischen Sprache welche durch die vorhandenen bzw. in Frage 6 vorgeschlagenen Visualisierungen noch nicht gelöst sind?

Reports

Reports werden verwendet um einen gewissen Aspekt eines Programms zu beleuchten. Ähnlich wie Visualisierungen sind Reports readonly, verwenden aber statt einer graphischen eine textuelle Notation. Die untenstehenden Reports werden bei den Fragen 1.3., 2.3., 3.3. bzw. 4.3. benötigt. Die Reports beziehen sich auf das unveränderte Programm **interview.mDML04**.

Report 1: textuelle Darstellung von Visualisierung 1 (**Frage 1.3.)**

```
Reachable States (DeviceState/UserLevel)
-------------------------------------------------------------
Pause/Remote (Initial), Pause/Manual, Measure/Remote,
Standby/Remote, Standby/Manuel


Transitions (DeviceState/UserLevel !Input! ->
DeviceState/UserLevel)
-------------------------------------------------------------
Pause/Manual !UserAction = SREM! -> Pause/Remote
Pause/Remote !UserAction = SMAN! -> Pause/Manual
Pause/Remote !UserAction = STBY! -> Standby/Remote
Pause/Remote !UserAction = SMES! -> Measure/Remote
Measure/Remote !UserAction = SPAU! -> Pause/Remote
Measure/Remote !UserAction = STBY! -> Standby/Remote
Standby/Remote !UserAction = SPAU! -> Pause/Remote
Standby/Remote !UserAction = SMES! -> Measure/Remote
Standby/Remote !UserAction = SMAN! -> Standby/Manual
Standby/Manual !UserAction = SREM! -> Standby/Remote
```

Report 2: textuelle Darstellung von Visualisierung 2 (**Frage 2.3.)**

```
DeviceState
-------------------------------------------------------------
Reachable States (DeviceState): Pause (Initial), Standby,
Measure,
Transitions (DeviceState -> !Input! -> DeviceState):
Pause !UserAction = STBY! -> Standby
Pause !UserAction = SMES! -> Measure
Standby !UserAction = SPAU! -> Pause
Standby !UserAction = SMES! -> Measure
Measure !UserAction = STBY! -> Standby
Measure !Useraction = SPAU! -> Pause


UserLevel
-------------------------------------------------------------
Reachable States (DeviceState): Remote (Initial), Manual
Transitions (DeviceState -> !Input! -> DeviceState):
Manual !UserAction = SREM! -> Remote
Remote !UserAction = SMAN! -> Manual
```

<u>Report 3: textuelle Darstellung von Visualisierung 3 (**Frage 3.3.)**</u>

Hier kann man etnweder Report 1 oder einen interaktiven Report verwenden.
Ein interaktiver Report könnte wie folgt aussehen (Konsoleninput/output):

```
Please enter state (DeviceState/UserLevel): Pause/Remote
Neighbors: Pause/Manual (!UserAction = SMAN!),
Standby/Remote(!UserAction = STBY!),  Measure/Remote
(!UserAction = SMES!)

Please enter state (DeviceState/UserLevel): Pause/Manual
Neighbors: Pause/Remote (!UserAction = SREM!)
```

<u>Report 4: textuelle Darstellung von Visualisierung 4 (**Frage 4.3.)**</u>

Auch hier benötigt man einen interaktiven Report (Konstoleninput/output):

```
Please enter state (DeviceState/UserLevel): Measure/Remote
Reachability check (y/n)?: y
Impasse check (y/n)?: y

Given state is reachable: Pause/Remote !UserAction = SMES! ->
Measure/Remote
Impasse unreachable from given state
```

Beispielprogramm

```
device Interview {
      public statevar DeviceState {Pause,Standby,Measure} = Pause;
      public statevar UserLevel {Manual,Remote} = Remote;
      input UserAction{SPAU,STBY,SMES,SMAN,SREM};

      given UserLevel = Manual when UserAction = SREM
            then UserLevel -> Remote;

      given UserLevel = Remote {
            given DeviceState != Measure when UserAction = SMAN
                  then UserLevel -> Manual;

            given DeviceState = Pause {
                  when UserAction = STBY then DeviceState -> Standby;
                  when UserAction = SMES then DeviceState -> Measure;
            }

            given DeviceState = Standby {
                  when UserAction = SPAU then DeviceState -> Pause;
                  when UserAction = SMES then DeviceState -> Measure;
            }

            given DeviceState = Measure {
                  when UserAction = SPAU then DeviceState -> Pause;
                  when UserAction = STBY then DeviceState -> Standby;
            }
      }
}
```

Bibliography

[1] Graphiti proposal. `http://eclipse.org/proposals/graphiti`. Accessed: 2016-11-20.

[2] Sirius gallery. `http://eclipse.org/sirius/gallery.html`. Accessed: 2016-11-20.

[3] Gail Anderson and Paul Anderson. *Essential JavaFX*. Pearson Education, 2009.

[4] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2013.

[5] Carolyn Boyce and Palena Neale. *Conducting in-depth interviews: A guide for designing and conducting in-depth interviews for evaluation input*. Pathfinder International Watertown, MA, 2006.

[6] Sabrina Bresciani and Martin J Eppler. Beyond knowledge visualization usability: toward a better understanding of business diagram adoption. In *2009 13th International Conference Information Visualisation*, pages 474–479. IEEE, 2009.

[7] Nick Cawthon and Andrew Vande Moere. The effect of aesthetic on the usability of data visualization. In *Proceedings of the 11th International Conference Information Visualization*, IV '07, pages 637–648, Washington, DC, USA, 2007. IEEE Computer Society.

[8] B. Cornelissen, A. Zaidman, and A. van Deursen. A controlled experiment for program comprehension through trace visualization. *IEEE Transactions on Software Engineering*, 37(3):341–355, May 2011.

[9] Fred D Davis. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS quarterly*, pages 319–340, 1989.

[10] Stephan Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[11] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.

[12] Mark Heckler, Gerrit Grunwald, José Pereda, Sean Phillips, and Carl Dea. *JavaFX 8: Introduction by Example*. Apress, 2014.

[13] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS Q.*, 28(1):75–105, March 2004.

[14] K. Hussein, E. Tilevich, I. I. Bukvic, and SooBeen Kim. Sonification design guidelines to enhance program comprehension. In *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, pages 120–129, May 2009.

[15] Jan Koehnlein. Eclipse diagram editors – the fxed generation (eclipsecon france 2014, toulouse). `https://www.youtube.com/watch?v=SiCYv3xgE6U`, 2014. Accessed: 2016-10-15.

[16] Jan Koehnlein. Diagrams, xtext and ux (eclipsecon na 2015, san francisco). `https://www.infoq.com/presentations/xtext-fxdiagram`, 2015. Accessed: 2016-10-15.

[17] Christian F. J. Lange and Michel R. V. Chaudron. Interactive views to improve the comprehension of uml models - an experimental validation. In *Proceedings of the 15th IEEE International Conference on Program Comprehension*, ICPC '07, pages 221–230, Washington, DC, USA, 2007. IEEE Computer Society.

[18] Salvatore Mamone. The ieee standard for software maintenance. *SIGSOFT Softw. Eng. Notes*, 19(1):75–76, January 1994.

[19] Tanja Mayerhofer, Philip Langer, Manuel Wimmer, and Gerti Kappel. xmof: Executable dsmls based on fuml. In *Software Language Engineering*, pages 56–75. Springer International Publishing, 2013.

[20] Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, and Philippe Vanderheyden. Eclipse development using the graphical editing framework and the eclipse modeling framework (2004). *IBM Redbooks*.

[21] Mark J Nelson. Game metrics without players: Strategies for understanding game artifacts. In *Artificial Intelligence in the Game Design Process*, 2011.

[22] Marian Petre and Ed de Quincey. A gentle overview of software visualisation. *Psychology of Programming Interest Group (PPIG)*, September 2006.

[23] Jasper Potts, Nancy Hildebrandt, Joni Gordon, and Cindy Castillo. Getting started with javafx. `https://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-architecture.htm`, 2014. Accessed: 2016-10-15.

[24] Dan Rubel, Jaime Wren, and Eric Clayberg. *The Eclipse Graphical Editing Framework (GEF)*. Addison-Wesley Professional, 2011.

[25] Z. Sharafi, A. Marchetto, A. Susi, G. Antoniol, and Y. G. Guéhéneuc. An empirical study on the efficiency of graphical vs. textual representations in requirements comprehension. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 33–42, May 2013.

[26] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings of the 1996 IEEE Symposium on Visual Languages*, VL '96, pages 336–, Washington, DC, USA, 1996. IEEE Computer Society.

[27] Juha Sorva, Ville Karavirta, and Lauri Malmi. A review of generic program visualization systems for introductory programming education. *Trans. Comput. Educ.*, 13(4):15:1–15:64, November 2013.

[28] M.-A.D. Storey, K. Wong, and H.A. Müller. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 36(2):183 – 207, 2000.

[29] Christian Stritzke and Sebastian Lehrig. Why and how we should use graphiti to implement pcm editors. In *Symposium on Software Performance – Joint Kieker/Palladio Days 2013*, CEUR Workshop Proceedings. CEUR-WS.org, November 2013.

[30] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[31] Markus Voelter, Thomas Stahl, Jorn Bettin, Arno Haase, and Simon Helsen. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2013.

[32] Richard Wettel, Michele Lanza, and Romain Robbes. Software systems as cities: A controlled experiment. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 551–560, New York, NY, USA, 2011. ACM.

[33] P. Young and M. Munro. Visualising software in virtual reality. In *Program Comprehension, 1998. IWPC '98. Proceedings., 6th International Workshop on*, pages 19–26, Jun 1998.