Visual Reflexion Models

Master's Thesis submitted to the

Faculty of Informatics of the *Università della Svizzera Italiana*

in partial fulfillment of the requirements for the degree of

Master of Science in Informatics

Software Design

presented by

Marcello Romanelli

under the supervision of

Prof. Michele Lanza

co-supervised by

Dr. Andrea Mocci

June 2014

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Marcello Romanelli
Lugano, 16 June 2014

To my beloved

Simplicity is about subtracting the obvious,
and adding the meaningful.

John Maeda

Abstract

Understanding a large software system is a complex task. To deal with this problem, one solution is to create a high-level model by abstracting from the source code entities. As a side effect, this will create a conceptual gap with the real underlying system. Thanks to a reflexion model it is possible to figure out if the high-level model is coherent or not with respect to the real system. In other words, a reflexion model allows to validate a given system against a developer's mental model.

Reflexion model entities are currently constructed from the repository of the system. Specifically, the only elements taken into account are the file system structure of the repository and the textual content of the source code files. Generating the source model by parsing files with an extraction tool is both time consuming and error prone since it requires the manual intervention of the user.
Another issue is that it is hard to understand the current coverage of the system: one cannot easily understand which source code entities are already mapped to high-level entities and which are not.

We believe that the idea of software reflexion model is powerful and we think that it can be of great benefit for anyone involved in writing software. This thesis presents our approach that starts from a meta-model of the source code and eliminates the step of encoding the high-level abstraction in favor of a purely visual selection. With the support of a web-based tool, we show how our idea can be effectively implemented in practice. The visual creation of abstractions allows the users to navigate the system as it is being analyzed and one eliminates the need of any "extra" language, reduces the possible errors and enables to understand if the final result is correct with respect to the initial mental model.

To validate our approach we apply it to two different case studies.

Acknowledgements

First of all I would like to express my gratitude to Prof. Dr. Michele Lanza who besides giving me advice and great ideas for this work, was able of convincing me, six years ago, in switching from Economics to Informatics with a great lecture about programming. The feelings of that day made me understand what I would do for the rest of my life.
I also want to thank Dr. Andrea Mocci, the co-supervisor of this work, for his guidance: without the many hours we spent together developing and discussing ideas, this work would not have been possible.
In general, I want to thank the entire *REVEAL* group for giving me the opportunity of being part of such an amazing and motivated research team.

My deepest gratitude goes to my parents, Lorella and Maurizio, for being the only ones who always believed in me since I was a kid. Their unconditional support and infinite love gave me the power to achieve amazing (and unexpected) results.

A special acknowledgment goes to my girlfriend Samira for always standing by my side. There are no words or numbers that can describe my gratitude and love for her. My only wish is spending the rest of our days together.

Last but not least, I want to thank all those people who thought that I would not have made it: without your words and actions I would never have found the grit and the stamina necessaries.

Contents

Contents xi

List of Figures xiii

List of Tables xv

1 Introduction 1

1.1 Contributions 2

1.2 Structure of the Document 3

2 Related Work 5

2.1 Common practices and tools 6

2.2 Modeling Techniques 6

2.2.1 Lightweight Modeling 7

2.2.2 Heavyweight Modeling 8

2.3 Reverse Architecting 8

2.4 Software Visualization 11

2.5 Summing Up 14

3 Reflexion Models 15

3.1 Introduction 15

3.2 Objectives 17

3.3 Architecture 18

3.3.1 High-level model definition 18

3.3.2 Source model extraction 19

3.3.3 Mapping definition 19

3.3.4 Reflexion model computation 19

3.3.5 Investigation and Refinement 20

3.4 Problems and Limitations 20

3.4.1 Source Model Generation 20

3.4.2 Coverage 21

3.4.3 Result Comparison 22

3.5 Summing Up 22

4 Visual Reflexion Models **23**
4.1 Source Model . . . 24
4.1.1 MSE . . . 24
4.1.2 FAMIX . . . 25
4.1.3 VerveineJ . . . 25
4.2 Backend . . . 25
4.2.1 Source Model Entities . . . 26
4.2.2 Entity Mapping . . . 26
4.2.3 Mapping Relations . . . 27
4.2.4 Reflexion Model computation . . . 28
4.3 Visualization . . . 29
4.3.1 Mapping . . . 30
4.3.2 Connection . . . 35
4.3.3 Analysis . . . 35
4.4 Summing Up . . . 39

5 Evaluation **41**
5.1 JHotDraw . . . 41
5.1.1 Model View Controller . . . 42
5.1.2 High-level model entities . . . 43
5.1.3 High-level model entities relations . . . 45
5.1.4 Reflexion Model computation . . . 46
5.1.5 Refinement . . . 46
5.2 ArgoUML . . . 51
5.2.1 High-level model entities . . . 51
5.2.2 High-level model entities relations . . . 55
5.2.3 Reflexion Model computation . . . 55
5.3 Reflections . . . 56
5.4 Summing Up . . . 56

6 Conclusions **57**
6.1 Summary . . . 57
6.2 Future Work . . . 58

Appendices **61**

A Architecture **63**
A.1 Backend . . . 64
A.2 Front-end . . . 65
A.3 Firebase (Storage) . . . 65
A.4 Relations . . . 66

B API **67**

Bibliography **73**

Figures

2.1 Lightweight sketch produced during a discussion 7
2.2 UML diagram of one part of our tool 8
2.3 The five levels of abstraction according to Riva 9
2.4 Reverse architecting process flow chart 10
2.5 Intensional View Editor 10
2.6 Visualization techniques. 12
2.7 Example of a Cone Tree visualization. 13
2.8 Three-dimensional Visualization techniques. 14

3.1 Models for the NetBSD virtual memory subsystem. 16
3.2 The reflexion model tool user interface. 17
3.3 Basic steps of the reflexion model approach. 18
3.4 Mapping between high-level model and source files. 21

4.1 Visual Reflexion Model contributions 23
4.2 FAMIX core essential entities. 25
4.3 Namespaces of JHotDraw. 26
4.4 Reflexion model's computation. 28
4.5 Reflexion model result. 29
4.6 Initial screen of our tool showing the four macro steps. 29
4.7 Definition of the MVC pattern on JHotDraw. 30
4.8 Tree diagram and corresponding Treemap. 30
4.9 Layouting algorithm for treemaps. 31
4.10 Interactive treemap layout. 31
4.11 Partial Tree representation of JHotDraw source model 32
4.12 Header of the treemap. 32
4.13 Treemap Layers. 32
4.14 Entities coloring. 33
4.15 Different views of the sidebar. 34
4.16 Dialog box showing saved mappings. 34
4.17 Top bar aspect when editing an high level entity. 34
4.18 Entity representation during linking phase. 35
4.19 Linking two entities. 35
4.20 Reflexion Model. 36
4.21 Relations between mappings in different high level model entities. 37
4.22 Inspection of a relation between two mappings. 38

5.1 Sample application built with JHotDraw. 41
5.2 M-V-C pattern implementations 42
5.3 UML diagram of JHotDraw. 43
5.4 Coverage of the high-level entity Model. 44
5.5 Coverage of the high-level entity View. 44
5.6 Coverage of the high-level entity Controller. 45
5.8 Links between MVC entities in JHotDraw. 45
5.9 Reflexion Model of JHotDraw 46
5.10 Refinement of JHotDraw's high-level model. 47
5.11 Reflexion Model of JHotDraw after the refinement 47
5.12 Inspection of the relations between Model and Controller 48
5.13 Inspection of the relations between `Controller` and `Model` 49
5.14 Inspection of the relations between `(.*)Tool` and `(.*)Figure` 49
5.15 Inspection of the relations between `Model` and `View` 50
5.16 Inspection of the relations between `(.*)Figure` and `(.*)Connector` 50
5.17 High-level architecture of ArgoUML. 51
5.18 UI subsystem of ArgoUML. 52
5.19 Coverage of the high-level entity UI. 52
5.20 Coverage of the high-level entity Design Critics. 53
5.21 Coverage of the high-level entity Code Generation. 53
5.22 Coverage of the high-level entity UML Meta Model. 54
5.23 Coverage of the high-level entity GEF. 54
5.25 Links between high-level model entities in ArgoUML. 55
5.26 Reflexion Model of ArgoUML. 55
5.27 Alternative architecture of ArgoUML. 56

A.1 High-level view of our tool. 63
A.2 High-level view of the backend. 64
A.3 Data representation in Firebase 65

Tables

4.1 Edges in the Reflexion Model 36

A.1 List of views 65

Chapter 1

Introduction

Understanding a software system is a complex activity [WY96]. There are cases where the complexity is due to an inconsistent or absent documentation and other cases where due to the complexity of software it is simply impossible to understand every detail. In similar situations, when we need to understand how a system works we first have to reverse engineer the various design artifacts to create something that is as independent as possible from the underlying implementation. As it is pointed out by Riva, with a reverse engineering process we move from lower to higher level of abstraction [Riv00]. The standard practice is that the developer builds an alternate view of the system by considering it from a higher abstraction level.

Abstractions are useful since they can largely ease the reasoning about software. On the down side, they can create a conceptual gap with the real underlying system. In other words, it might happen that what is described with the use of an abstraction is not corresponding to the concrete artifact behind it. More precisely, Bowman et al. made a distinction between the conceptual architecture and the concrete architecture of a system [BHB99]. The conceptual architecture is the way developers think about a system, while the concrete architecture of a system shows the actual relationships that exist in the implemented system. With the purpose of understanding where and how a given system and its high-level representation differ, Murphy et al. proposed the concept of *reflexion models* [MNS01]. Conceptually, reflexion models allow developers to express the mental model they have about a software system and compare it to the actual architecture. This technique can be used for a variety of tasks: from checking whether the system a developer is reasoning about is the same as the real system to the identification of possible design problems.

The approach proposed by Murphy et al. included five basic steps [MNS01]. (1) First, the developer specifies a high level model of the system as it is present in his own mind. (2) Next, he extracts the source model from the artifacts of the system by taking into account the structure of the repository in the file system and the textual content of the source code. (3) In the third step, the user describes a mapping between the extracted source model and the stated high-level structural model. (4) This first three steps are used as an input for the fourth step where the reflexion model is computed. (5) The last step is about the interpretation of the reflexion model in order to derive information that will helps the engineer reasoning about the system. In a study conducted by Murphy et al., it is shown how the idea of reflexion models can be successfully applied to very complex system [MNS01]. More precisely, they ask a software engineer at Microsoft to use their tool to generate a model of Excel. The

result of this experiment is particularly meaningful since the developer reports that in two weeks time he was able to understand concepts of the system that with other approaches would have take more than 2 years. Despite those promising results, we think that the original reflexion model approach has three major weaknesses. (1) First, generating the source model by parsing the various files with an extraction tool is both time consuming and error prone given that it requires the manual intervention of the user. Specifically, the source model depends on concepts that are strictly related to specific programming languages. For example, there is a huge difference between a system written in Java and a system written in C++: in the former case information such as the package organization can be inferred from the structure of the repository while in the latter case we need to perform some parsing of the code. (2) Next, we do not have any metric that allows the developer to understand how much of the actual system is being covered by the high-level model that he has generated. Last but not least, this approach may leads to results that are difficult, or even impossible, to compare since it is up to the developer to select a tool for the extraction of the source model.

We believe that the idea of software reflexion model is powerful and we think that it can be of great benefit for anyone involved in writing software, from undergraduate students to professional engineers. In this thesis we present a revision of the classical reflexion model approach. The first change is in the way in which we create the source model: instead of starting from the system artifacts we start from a FAMIX meta-model of the system which makes the whole approach more powerful and flexible. Thanks to this change we have a richer source model that allow the user to be more expressive in the mapping they can create with the stated high-level structural model. Additionally, we introduced a visual representation of the system that allow the user to navigate the system and show how the high-level model maps to the source model. Besides revisiting the reflexion model approach, we created a web based application that illustrates how our idea can be effectively putted into practice. The main focus of our application is the ease of use and simplicity, we want everything to be visual so that we limit the possible errors and we virtually enable anyone to understand if the final outcome corresponds to the initial mental model. With our approach, reflexion models become easier to construct, and thus applicable to a wider range of cases: from the analysis of big projects that involve professional software engineers, to small projects managed by undergraduate students.

To validate our approach we analyze the application of our approach to two different case studies. The first system we analyze is JHotDraw since it heavily relies on some well-known design patterns. Next we consider a big project such as ArgoUML (more than 100'000 lines of code in over 800 classes). The results obtained give us confidence of the usefulness of our ideas.

1.1 Contributions

The contributions of this thesis can be summarized as follows:

- An intuitive visual approach for building, analyzing, and refining reflexion models.
- The implementation of a web-based application to help software engineers align their mental vision with an existing software system.
- A stand-alone plugin to visualize meta-models with an interactive squarified treemap.

1.2 Structure of the Document

The rest of the document is structured in different chapters, described in the following.

- **Chapter 2** gives an overview of the research areas related to our work and presents the projects which have influenced, at least in part, our choices.
- **Chapter 3** introduces the reader to the software reflexion model approach proposed by Murphy et al. with a real world example [MNS01]. After the example we explain the objectives, architecture and shortcomings of this technique.
- **Chapter 4** describes our contribution, the Visual Reflexion Model approach, which an evolution of the original reflexion model technique that allows users to create and analyze visual abstraction with ease. We also present the tool we created to fully support our modeling technique.
- **Chapter 5** provides an evaluation of the Visual Reflexion Model approach by means of two case studies: JHotDraw and ArgoUML. Specifically, we start by describing the methodology used for the evaluation, then we illustrate in detail each case study and finally we discuss the results we obtained.
- **Chapter 6** summarizes and concludes this thesis by going over the contributions of our work. It also illustrates what is the future works that we plan to investigate to improve our tool.

Chapter 2

Related Work

The vast majority of people interact with software only at the surface level, by entering some inputs and waiting for a response. What is exposed to the end users is usually only the facade of a much more complex system. When a software engineer needs to perform some task on a system, he is required to have a deep understanding of it. In order to gain insight, there are a number of approaches that are different in terms of time required, usefulness and applicability. However, a research carried out by Letovsky suggests that there is not a single cognitive model that can explain the behavior of the various programmers, and that it is more likely that programmers, depending on the particular problem, will alternate between different kinds of models [Let86].

Software comprehension has been defined by Deimel and Naveda as "the process of taking computer source code and, in some way, coming to understand it" [DN90]. Müller et al. defines software comprehension as "the task of building mental models of the underlying software at various abstraction levels, ranging from models of the code itself, to models of the underlying application domain, for maintenance, evolution, and reengineering purposes" [MTW93]. Woods et al. summarize the process of program understanding as "the process of making sense of a complex source code" and they focus on formally proving that it is an NP-hard problem [WY96].

The reflexion model approach proposed by Murphy et al., enables **software comprehension** through a **reverse architecting** process [MNS01]. Specifically, the reflexion model approach supports the developers on understanding a software system by means of the creation of a high-level architecture that will be then compared to the actual system.

In the following sections we explain which are the techniques used by developers to understand software systems (Section 2.1) and how their models are graphically represented (Section 2.2). We then give an overview of related approaches to the reflexion model technique which include the research areas of reverse architecting (Section 2.3) and software visualization (Section 2.4).

2.1 Common practices and tools

When programmers need to comprehend a software system, they tend to use a bottom-up or a top-down approach. The bottom-up approach is generally used in situations where the programmer is unfamiliar with the domain and uses the source code as a starting point. Reading the source-code, allows abstract concepts to be formed by chunking together low-level information. In other words, the developer starts from the source code and then mentally chunks or groups lines of code into higher-level abstractions. For example, in a web application you can have multiple classes that take care of reading the inputs by the users and outputting some sort of response. In this case, since those entities exhibit a similar behavior, we can decide to aggregate them into a single higher level entity named `Controller`.

In contrast, top-down approaches are better used when the programmer already has some knowledge of the domain. This kind of approaches are based on the idea that the programmer utilizes his knowledge to build abstractions that are later mapped to lower level artifacts such as the source code. The two previous approaches are not excluding each other. As von Mayrhauser et al. point out, developers adopt a predominant strategy based on their domain knowledge and switch between the two approaches as cues become available to them [vMVH97].
O'Brien suggests that every software comprehension model has four common elements [O'B03]:

1. **Knowledge base**: knowledge of the programmer prior to the understanding of the code. It may include understanding of the domain, general information of the domain, common standards and practices. The knowledge base expands as the level of programmer understanding increases.
2. **Mental model**: programmer's existing or current representation of the system under analysis.
3. **External Representation**: anything that helps the programmer comprehending code. This can be system documentation, source code, expert advice from other programmers familiar with the problem domain, or any other source code similar to the code under observation.
4. **Assimilation Process**: the strategy used by the programmer to comprehend the source code, and is responsible for updating and augmenting the programmer's mental model.

The previous elements allow the developer to formulate hypotheses about the system under analysis. Brooks shows that during the code comprehension phase programmers use an iterative, hierarchical method of generating, refining, or updating the initial hypotheses [Bro83].
During the assimilation process is a common practice for software engineers to reason about a system by creating graphical models of the system under analysis. In the following section, we explain the different kinds of modeling techniques available.

2.2 Modeling Techniques

Modeling Techniques can be split in two categories: informal (lightweight) and formal (heavyweight) methodologies. Intuitively, with an informal methodology developers does not have strict rules to follow and thus they can freely express their ideas. On the other hand, a heavyweight methodology expects developers to apply the rules rigorously so that the output can be understood and exchanged between people.

2.2.1 Lightweight Modeling

Cherubini et. al conducted a study where they showed that when developers reason or want to share their ideas with others, they tend to sketch potential solutions using lightweight, informal, means such as pen and paper or whiteboard drawings [CVDK07]. The main reason behind this traditional approach is that current tools are not capable of helping developers externalizing their mental models of the code. Additionally, a lightweight technique, according to Petre, is at the base of the creative environment required in software modeling and essential to enable the exploration of new design solutions [Pet09]. Sketching is one of the most natural way of externalize design thinking, thanks to its flexibility and its ease of adapting to different focuses. Being able to change focus, as reported by Petre, helps the developer reason across the full breadth and depth of software designs in order to consider consequences and implications of design decisions [Pet09]. Additionally, as Dekel et al. point out, lightweight modeling allows the developer to choose how each part of the model should be specified [DH07]. Specifically, a developer can decide to use a combination of different approaches, such as a mix of formal and informal notations, to potentially create new ones (Figure 2.1). Last but not least, this approach does not require any technical skills making it well suitable for an immediate use.

Despite having many advantages and being largely diffused, this approach has two major deficiencies: First, the output originated from a design session is in most cases static and thus, especially when using physical mediums, it is hard to manipulate it without having to erase parts or having to re-draw them. The need of manipulating the design, as described by Dekel, is an essential feature since software engineers often needs to rethink and reorganize the various parts in a more complex fashion [DH07].

Second, if the modeling happens solely on physical mediums, it is harder to store, share and reuse the created artifacts. Indeed, such representations are of great benefit for the insights they gave during their usage but after that moment their usefulness decrease with an exponential decay.

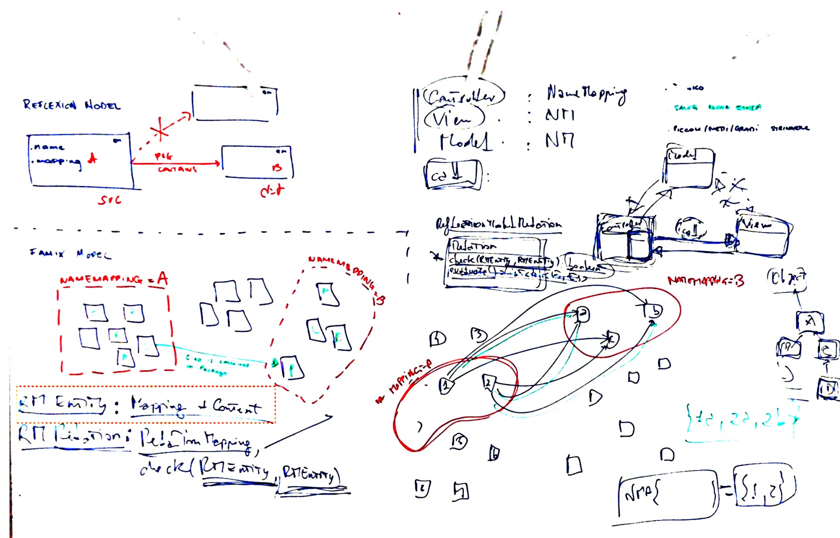


Figure 2.1. Lightweight sketch produced during a discussion

2.2.2 Heavyweight Modeling

To overcome the problems of lightweight modeling explained in Section 2.2.1, developers adopt a different strategy. Among the various heavyweight approaches, UML (Figure 2.2), the unified modeling language, according to Berardi et al. is the de-facto standard formalism for software design and analysis [Ber05]. The most obvious advantages of using an heavyweight modeling language such as UML is an improved visual consistency thanks to the fact that the developer is not required to use his drawing skills. As a consequence of the application of a standardized approach, diagrams acquire an uniform style, increasing the general understandability. Additionally, such diagrams can be easily stored and thus can be shared among developers.

Although heavyweight modeling solves some of the shortcomings of a lightweight approach, it carries a number of limitations. For instance, the strict rules and the formalism imposed, limits the creative approach of the developer. Indeed, the main goal of UML is creating diagrams that are visually clean and easy understandable rather than enhancing the discovery and evaluation of different design solutions. Despite using a standardized and common notation, UML diagrams are saved using different formats depending on the UML editor being used. Having different formats makes them not suitable for a seamless sharing. Lastly, Berardi et al. shows that reasoning on UML diagrams is EXPTIME-hard [Ber05]. More informally, we can say that this approach requires the modeler to have a knowledge of the standard used that, in the case of UML, can take up to several years of practice and study.

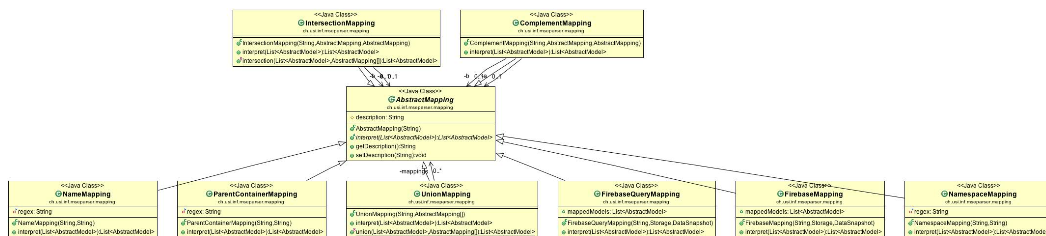


Figure 2.2. UML diagram of one part of our tool

Both lightweight and heavyweight modeling techniques are employed to create a visual representation of a system from an high-level perspective. This perspective is often referred as the *architecture of the system* and in the next section we explain what are the major approaches to recover it from of an existing software.

2.3 Reverse Architecting

As Krikhaar stated, “Reverse architecting is a flavor of reverse engineering that concerns all activities for making existing (software) architectures explicit” [Kri97]. In general the main goal of reverse engineering is to increase comprehensibility of the system either for maintenance tasks or for new development [Riv00]. Below we summarize the contributions that we think are the most important and influenced the most our work.

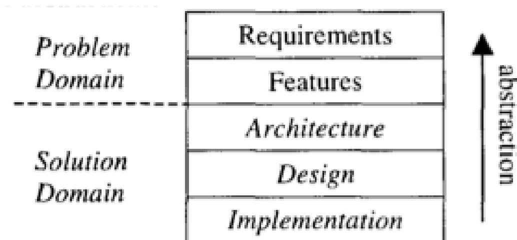


Figure 2.3. The five levels of abstraction according to Riva

In the approach to reverse architecting proposed by Riva, the analysis starts at the implementation level and moves towards the architecture level (Figure 2.3) [Riv00]. The goal of this process is to extract the architectural models of a software system starting from its implementation. Bowman et al. call those two models respectively concrete architecture and conceptual architecture [BHB99]. In the first case they “show the relationships that exist in the implemented system” while in the latter they “show how developers think about the system”. The approach created by Riva (Figure 2.4) aims at assisting the activities of software architecture recovery and definition of architecture improvement plans trough six macro-phases [Riv00]:

1. *Definition of architectural concepts*: definition of the building elements of a system. Such elements can be for example subsystems, modules and classes. Those elements, can be identified by looking at the documentation of the system, at the program source or by interviewing experts. Unfortunately, architectural description of existing system are rare, and thus these concepts have to be extrapolated from different artifacts. Next, the developer needs to understand how the architectural concepts he identified previously are represented in the source code.
2. *Extraction of the source code model*: the most trustworthy artifact for creating an architectural description of the system is, of course, the source code of the software system being analyzed. Therefore in this approach it is analyzed to collect all the relevant architectural information. The architectural concepts identified in the previous step are essential in this phase since they represent the types of the entities that will populate the source code model.
3. *Abstraction*: the source code model per se is only a low level view of the system as extracted from the source files. In this phase, the developer needs to create architectural abstractions by grouping together entities he thinks are belonging to the same high-level concept. The abstractions identified can be of two different types: known and unknown. In the first case they are documented and have been explicitly used in the design of the system. In the second case, they emerged from the evolution of the system.
4. *Improvement of architecture documents*: the documents are re-written and extended in order to increase the understanding of the actual (concrete) architecture of the system.
5. *Analysis of extracted architecture*: the goal of this analysis is to identifying the architectural shortcomings and possible improvements for the system.
6. *Architectural reorganization of source code*: the system is modified according to the solutions identified in the previous step.

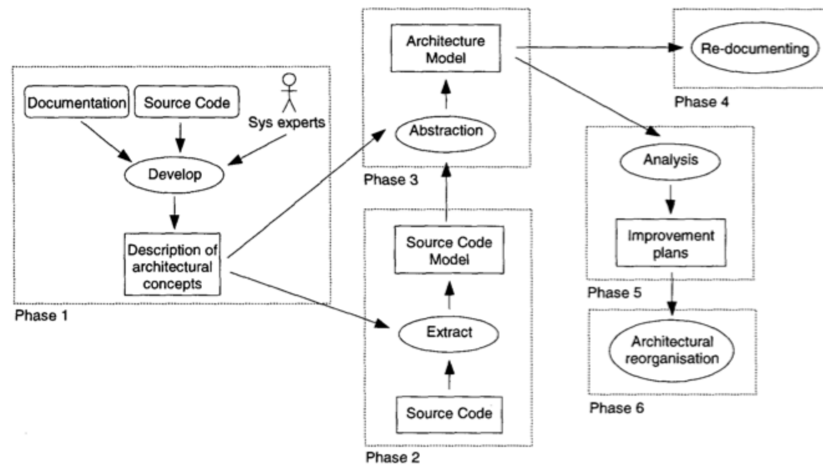


Figure 2.4. Reverse architecting process flow chart

In the work developed by Mens et al., authors propose a set of tool for the documentation and co-evolution of high-level structural regularities in the source code of a software system [MKPW05]. Among the set of tools developed, the *Intensional View Editor* is particularly meaningful. An intensional view is intended as a set of source-code entities (classes or methods) which share a common structure. Groups of elements, referred in this context as views, are defined by means of intensions. An intention is an executable description that when evaluated produces the set of entities belonging to that particular view. The set produced by the execution of the intension is also called the extension of the view. The Intensional View Editor also allows the user to deal with deviating cases in the source code thanks to the explicit exclusion or inclusion of an entity from a view.

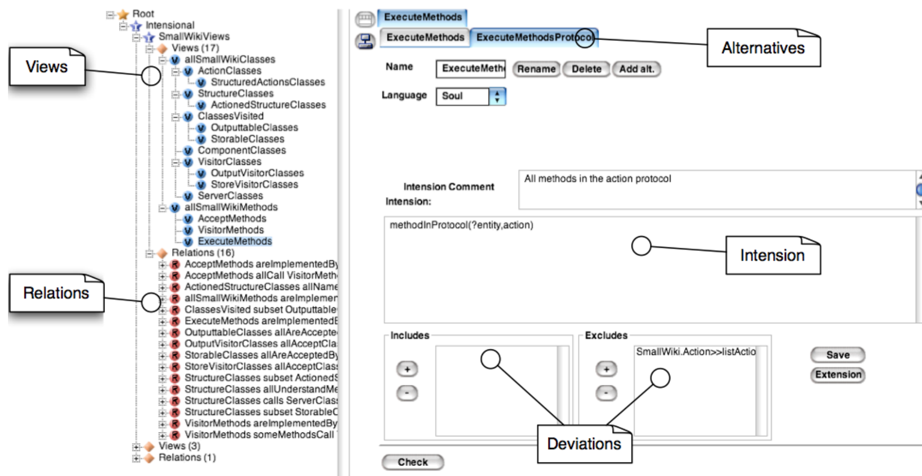


Figure 2.5. Intensional View Editor

Another valuable tool is the *Relation Editor* which allows a user to document relations between

intensional views. Relations can be expressed in the canonical form:

$$Q_1 x \in \texttt{Source} \; : \; Q_2 x \in \texttt{Target} \; : \; x \; R \; y$$

where Q_1 and Q_2 are either mathematical logic quantifiers $\forall, \exists, \ldots$ or custom quantifiers. This means that these custom quantifiers can be something like *some*, *few*, *many* or *most* where each one of them is defined in terms of a minimum or maximum number of elements for which the condition is valid. *Source* and *Target* are intensional views and R is a binary predicate over the source-code entities contained in the respective views.

During the architecture recovery process it is essential for the developers to construct the "big picture" by making sense of all the available artifacts. In the specific case of the source code, it is possible to gain additional insights by considering it from a visual perspective. In the next section we illustrate different software visualization techniques developers can adopt.

2.4 Software Visualization

As Ball and Eick pointed out, software systems are invisible by their nature and this contributes to hiding their complexity [BE96]. More precisely software is intangible, having no physical shape or size. With software visualization tools, developers use graphical techniques to make the software visible again. Such information can include metrics about the programs, artifacts, and behavior. In general, visual representations aim at simplifying the process of program understanding. Thanks to the power of pictures, knowledge is preserved both helping existing developers to remember and new members to discover how the code works.

Young and Munro propose a list of desirable properties for visualization involving software [YM98]:

Simple navigation with minimum disorientation: the visualization should have a structure and should include elements to helps the user in the navigation task.

High information content: the visualization should include as much information as possible without overwhelming the user. This two goals are of course in contradiction and thus the real aim is to find an acceptable balance.

Low visual complexity, should be well structured: in order to provide an easy navigation information should be structured in a correct way.

Varying levels of detail: the level of information provided should change according to the user's will. For example, when a user first enters the visualization he expects to visualize the system in its entirety. As they gain confidence with the system users should be able to investigating areas of interest in greater detail.

Resilience to change: if the information upon which the visualization is built is slightly modified or the interest of the user shifts the visualization should not have major transformations.

Good use of visual metaphors: visual metaphors that are familiar to the users provides a good starting point for gaining an understanding about the visualization.

Approachable user interface: the user interface should accommodate needs and be intuitive, without introducing unnecessary overheads.

Integration with other information sources: the visualization should be linked with the original information it represents (e.g. the source code)

Good use of interaction: by allowing users to interact with the visualization it is possible to gain more information and also helps to maintain interest.

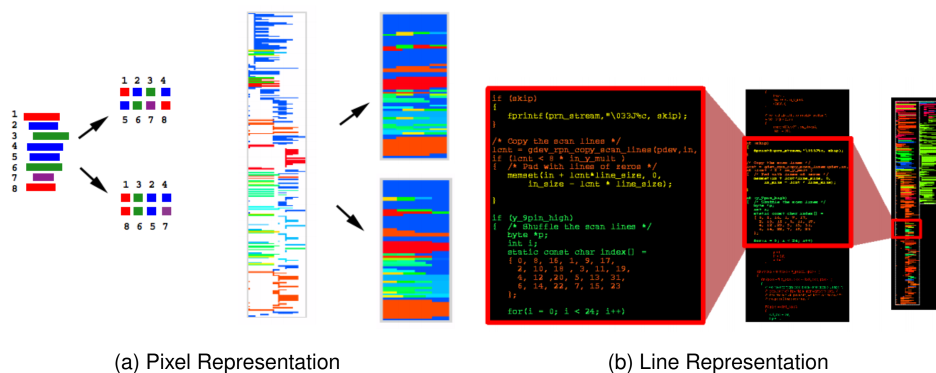
Suitability for automation: the visualization should be created as much as possible in an automatic manner in order to be applicable to indefinitely large cases. maintain interest.

It is important to notice that many of these qualities are mutually exclusive and therefore it is necessary to make compromises.

Generally speaking, software visualization techniques ranges from two-dimensional structures to more evolved three-dimensional representation and, more recently, also to virtual environment.
Two-dimensional approaches usually involve graph or treelike representations. However, when the number of elements considered starts to become large, the graph is sub-divided in different views or windows so that the user can focus on the level of detail he desires.
Ball and Eick devised a number of two-dimensional techniques that can be used to visualize software [BE96]. For example, we identified the following interesting ones:

Line Representation Figure 2.6a illustrates the line representation of a source code file at varying scales. The aim of this technique is visualizing large pieces of code without necessarily having to read the actual text. Indeed, if we assume standard coding conventions such as the indentation of loops and conditional statements, with this kind of visualization it becomes easy to spot the program's structure.

Pixel Representation In this visualization (Figure 2.6b) each line of code is encoded using a small number of color-coded pixels. This allows to achieve a higher information density compared to the line representation. The pixels are ordered left to right in rows within the columns, each column corresponding to a single file. With this kind of representation it is possible to show over a million lines of code using a standard display. The size of the various rectangle are directly related t the file size, and thus finding files based on their size becomes a relatively easy task.



(a) Pixel Representation (b) Line Representation

Figure 2.6. Visualization techniques.

Unfortunately these kind of techniques may lead to having a large amount of information confined to a flat plane that as a consequence will result in a cluttered visualization. Indeed, Stasko and Wehrli

suggest that “by adding an extra spatial dimensions, we supply visualization designers with one more possibility for describing some aspect of a program or system” [SW93]. A first experiment in this direction, is the Cone Tree (Figure 2.7) developed by Ware et al.: with this kind of visualization the creators claim that it is possible to visualize up to 1000 nodes without incurring in visual cluttering [WHF93]. Generally speaking, a three-dimensional visualization enable the users to perceive the depth of a presented structure. More precisely, whit a 3D visualization user can spatially navigate the structure by choosing different angle of views that in some cases may reveal structures that were previously hidden. The use of three-dimensional representations increases the sense of familiarity and realism with a system [KM99]. In fact, the world in which we live is (as far as we know) a three-dimensional experience and therefore visualizing a software system in a way close to reality reduces the cognitive strain on the user.

Wettel et al. propose a three-dimensional visualization tool (Figure 2.8a) which supports software analysis tasks [WLR11]. Their approach makes use of the familiar metaphor of a city to depict a software system. Precisely, a software system is being represented as a city that is made of buildings that can have various shapes. Those buildings represent classes and the various districts represent the packages in which they are contained. Software metrics are mapped to the various city artifacts. For example, the number of methods in a class is mapped on the height of a building while the number of attributes determines both the width and length of the base.

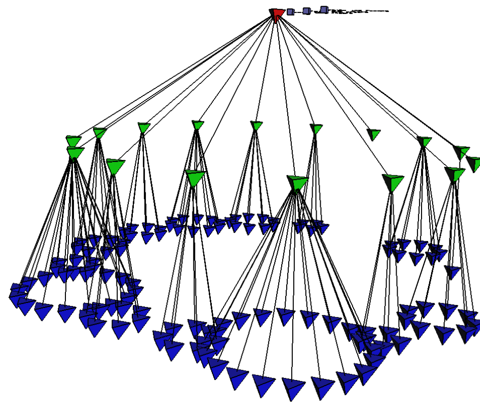
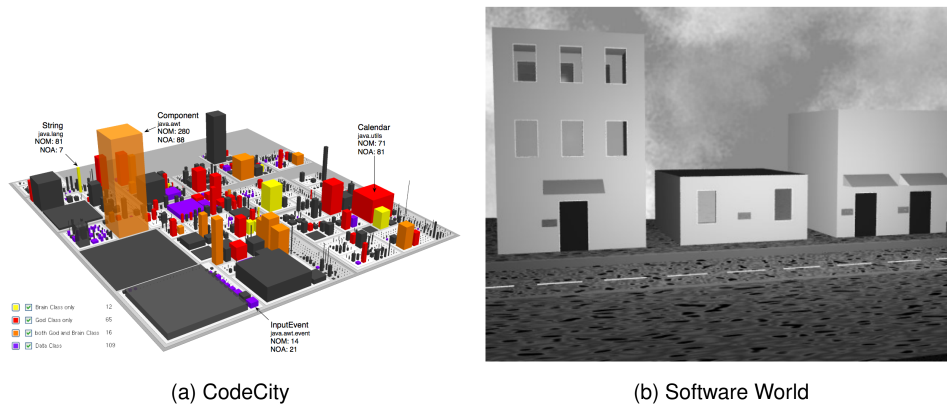


Figure 2.7. Example of a Cone Tree visualization.

In contrast with three-dimensional visualizations, Virtual Environments allow the user to became part of the representation of the system. During his “immersive experience”, the user can take full advantage of his stereoscopic vision that will help in disambiguating complex abstract representations. Moreover, being part of the environment under analysis will allow the user to easily judge relative size of objects and distances between objects. This is harder in a classical three-dimensional visualization because the user needs to change the viewing angle in order to understand the diagram.
An example of a tool that uses virtual environments to explore a software system is shown in the research carried out by Knight and Munro [KM99]. In their approach the various parts of Java code are mapped to real-world metaphors like the world, countries, districts, and buildings (Figure 2.8b).



(a) CodeCity (b) Software World

Figure 2.8. Three-dimensional Visualization techniques.

2.5 Summing Up

In the previous sections we gave an overview of the research areas related to our work and we discussed projects which have influenced, at least in part, our choices. We started this chapter with a section dedicated to the methods used by developers to tackle the software comprehension problem. Next, we outlined the notion of *software modeling* by explaining both lightweight, informal, methodologies as well as heavyweight, formal, approaches. We then explained how developers *reverse engineer the architecture* of an existing system by pointing out two meaningful studies that shaped our work. Last but not least, we defined the concept of *software visualization* by providing an overview of the state of the art related to this field.

In Chapter 3, we present in details the reflexion model approach by illustrating the general architecture, the objectives and the shortcomings of this technique.

Chapter 3

Reflexion Models

In this chapter we introduce the reflexion model approach as proposed by Murphy et. al [MNS01]. More precisely, we start by giving an initial overview of the approach together with an introductory example (Section 3.1), we move to the objectives (Section 3.2) and then we explain the overall architecture (Section 3.3). Last but not least, we expose what we think are the shortcomings of this technique (Section 3.4).

3.1 Introduction

In essence, the reflexion model technique can help a developer understanding where one artifact (such as his mental model about the software system) is consistent or inconsistent with another artifact (such as the source code). Before moving on to a concrete example, we introduce the terminology used in the reflexion model approach:

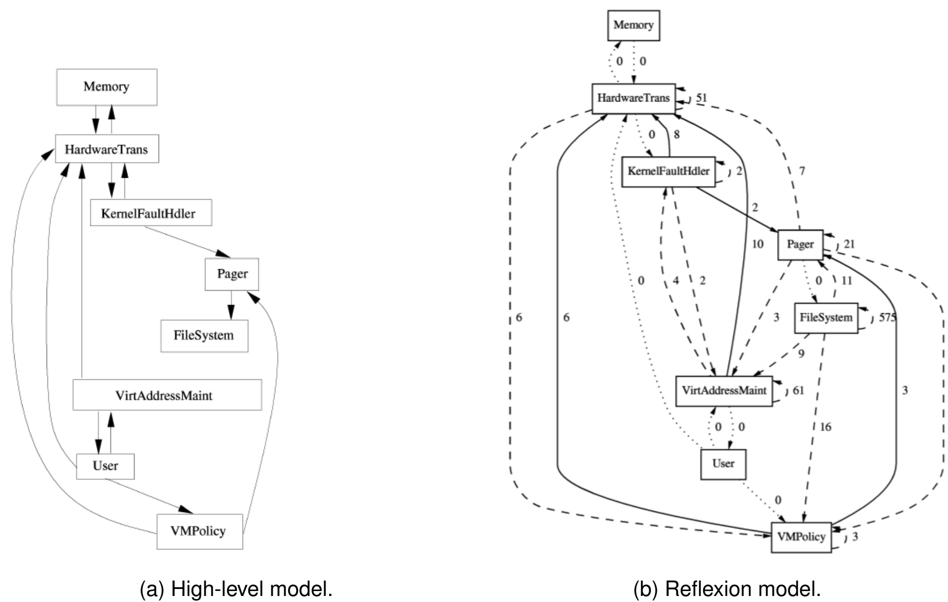
Source model is a representation of the system under analysis. In other words, it represents the current structure of the system. A source model can vary depending on the underlying language and it can include entities like namespaces, classes, methods, calls and so on.

High-level model is the representation of the system under analysis as it is present in the developer's mind. This kind of representation comprises only two elements: modules (high level abstractions) and relations between them.

Mapping is a named collection of source model entities. These collections can be created essentially either using physical (e.g. directory and file) or logical (e.g. functions and classes) attributes.

Arc is a relation between two entities not necessarily different. Arcs can be of different types: convergent, divergent and absent.

As an introductory example Murphy et al. describe how a developer with expertise in Unix virtual memory (VM) systems used the reflexion model technique to gain understanding of an unfamiliar implementation, NetBSD [MNS01]. NetBSD is a complex software that comprises about 250000 lines of C source code distributed over approximately 1900 different files. Such a system is therefore too large to be examined directly (i.e. by inspecting the source code) and, as it happens in many other software systems, appropriate documentation and existing expertise cannot be exploited easily. To apply the reflexion model technique, the developer iteratively performed five basic steps shown in



(a) High-level model.

(b) Reflexion model.

Figure 3.1. Models for the NetBSD virtual memory subsystem.

Figure 3.3. In the first step, the developer created a mental representation of the system (Figure 3.1a). This model, that was purely based on the developer's beliefs, contained the different modules of a virtual memory implementation and the calls among them. Next, the developer extracted the source model (in this case calls relation between NetBSD functions) from the artifacts of the system. This output was then translated using a small awk script in order to be used as an input for the software reflexion model tools. By the end of this step, the extracted calls relation comprised over 15000 entries describing calls between over 3000 source entities. In the third step, the developer specified a mapping between the extracted source model and the stated high-level model (see Listing 3.1 for the full mapping).

Listing 3.1. Reflexion Model mapping for NetBSD

```
1 [file=.*pager.*    mapTo=Pager ]
2 [file=vm_map.*     mapTo=VirtAddressMaint ]
3 [file=vm_fault\.c  mapTo=KernelFaultHdler ]
4 [dir=[un]fs        mapTo=FileSystem ]
5 [dir=sparc/mem.*   mapTo=Memory ]
6 [file=pmap.*       mapTo=HardwareTrans ]
```

After completing the previous three steps, the developer, in the fourth step, used a tool to compute the reflexion model (Figure 3.1b). The output of this computation was a comparison between the source model and the high-level model specified by the developer. Specifically, in this case, we can see that the developer correctly predicted interactions between functions in modules implementing `VMPolicy` and functions in modules implementing `Pager`. On the other hand we have situations where the idea of the developer was not aligned with the actual system. Considering for example the relation from `FileSystem` to `Pager`: we have source interactions that were not expected by the developer, in other words this relation was not present in the high level model. We have a different situation for the relation

going from `Pager` to `FileSystem`: in this case the relation was specified by the developer, however in the source model those two entities were not related. In the fifth and last step, the developer analyzed the reflexion model to derive information that helped him in the understanding process. For example, the developer can try to understand what are the reason of the relation between `FileSystem` and `Pager` that he omitted from his high-level model. In Figure 3.2 it is shown a window that displays the result of the developer inspecting the arc between `FileSystem` and `Pager`.

Figure 3.2. The reflexion model tool user interface.

3.2 Objectives

When a software engineer needs to perform some tasks on an existing system he must have some understanding of the system's source code. However, when the system considered starts to become large, gaining insight into the source code typically is too time consuming and expensive. The reflexion model approach proposed by Murphy et al. aims at allowing software engineers to rapidly and cost-effectively gain a good-enough knowledge about a system's source code [MNS01].

To prove the applicability of their approach, Murphy and Notkin decided to test it by applying it to large systems under the constraints of an industrial environment [MN97]. At the time of their research, there was a group at Microsoft that was trying to performing an experimental reengineering of Microsoft Excel. More precisely, they needed to locate and extract components from the source code. As a prerequisite to this activity, however, the team needed to understand some of the structure of the 1.2 million lines of C code. One of the engineers in the team (with more than 10 years of experience) decided to apply the reflexion model technique to understand how Excel's C source code divides into static modules and how those modules interact at execution. Traditionally, to understand the logic and the architecture of the system, the engineer would have to read a document called "Excel Internals" or he would rely on verbal explanations or even study directly the source code. All the previous three approaches were usually applied successfully. However, in the specific case of experimental reengineering, they were not appropriate both because of time constraints but also because the mentor from the Excel development group could not guarantee his full availability through the whole process. After the first day using the approach proposed by Murphy et al., the engineer produced the first initial reflexion model and then he spent another 4 weeks refining it [MNS01]. By the end of this process the engineer estimated that gaining the same degree of familiarity with the Excel source code might have taken up to two years with other available approaches.

3.3 Architecture

The reflexion model approach starts with the definition of an high-level model by the users. The model defined can be seen as a system abstraction tailored to the desired software engineering task. Most of software engineers, as illustrated by Cherubini et al., use an informal structural model to reason about systems [CVDK07]. However, this way of reasoning implies a significant risk because the model depicted is totally disconnected from the source. The next steps of this technique are to extract a source-model from the system's artifacts, define mappings between the source-model and the high-level model, and, through a set of computation tools, compare the two models. This enables software engineers effectively check their mental-model with the information present in the source code. More precisely, as it is shown in Figure 3.3, to derive a software reflexion model and iteratively refine it, the user must performs five basic steps:

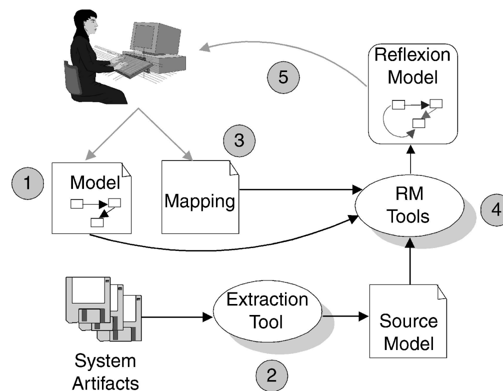


Figure 3.3. Basic steps of the reflexion model approach.

1. High-level model definition
2. Source model extraction
3. Mapping definition
4. Reflexion model computation
5. Investigation and iterative refinement

We will now explain each one of them in details.

3.3.1 High-level model definition

The high-level model describes conceptual aspects of the system's structure. To describe such a model an engineer needs to acquire as much information as possible about a given system. Information can be found by reviewing artifacts (source code and documents), by interviewing experts, or by looking at similar architectures. In other words, the engineer should consider anything that may give information about the system. For instance in the NetBSD example, the developer defines 8 high-level model entities (Figure 3.1a) namely: `Memory`, `HardwareTrans`, `KernelFaultHdler`, `Pager`, `FileSystem`, `VirtAddressMaint`, `User` and `VMPolicy`.

3.3.2 Source model extraction

Extracting the source mode does not require the engineer to reason about a system. Indeed, it is possible to use an automated tool that generates a source model by considering the interaction information extracted either from the call graph or from the dependencies between the various files that compose the system. The tool used to extract the source model, which is not given as a part of the approach, is selected (or created) by the user. For this specific reason, the output may be different from tool to tool. As a direct consequence, the user may need an additional piece of code that given the output produced by the extraction tool it will transform it in a format that can be read by the software reflexion model tools. In the introductory example, the engineer used the *xrefdb cross-reference database tool* (developed by Reiss [Rei95]) to extract calls relation between NetBSD functions. However, before it could be used, the developer wrote a small awk script to translate the output of *xrefdb* into the input format expected by the software reflexion model tools.

3.3.3 Mapping definition

At this point the user defines a mapping between the entities in the source and the entities in the high-level model. To facilitate the mapping process, a user may refer to the information contained in the source model that can be either physical (e.g. files and directories contained in the repository) or logical (e.g. classes and methods) software structures. Additionally, the user may also make use of regular expressions so that the specification of the mapping becomes more concise. For example, a user can map to the entity `Model` every file whose name contains the sub-string `Figure` with the following line of code: `[file=.*Figure.* mapTo=Model]`. Alternatively, a similar mapping can be created by referring to the methods contained in the source model. The snippet of Listing 3.1 represents the actual mapping created for the NetBSD example. The structure of every line is very simple: on the left side we have the declaration of the source model entity while on the right we have the name of the high-level entity it maps to. For instance, in the first line it is declared that every file in the system which name contains the string *pager* is going to be mapped to the high-level entity named *Pager*.

3.3.4 Reflexion model computation

The modeler can use as input the high-level model, the source model, and the map to invoke a set of tools that will compute and display the software reflexion model. The computation of the reflexion model consists of pushing each interaction present in the source model through the map to form induced arcs between high-level model entities. Next, the induced arcs and the interactions stated in the high-level model are compared to produce a reflexion model. The outcome allows the user to see what are the interactions in the source code from the viewpoint of the high-level model. More precisely the view (Figure 3.1b) consists of the entities defined in the high-level model and three different kinds of arcs: convergent, divergent and absent. In the first case the arcs represent mapped interactions that agree with the stated high-level model, in the second case they represent mapped interactions not stated in the high-level model and in the last case they represent interactions stated in the high-level model that do not correspond to any mapped interaction. In addition, each arc has an associated number that shows how many source model interactions are mapped to it.

3.3.5 Investigation and Refinement

During the interpretation of the results, the developer may either decide to refine one or more of the inputs (source model, high-level model or map) and compute a new reflexion model, or else he may decide that he has acquired the desired knowledge. However, in general, viewing the result of a reflexion model computation (e.g. Figure 3.1b) does not usually provides enough information for a user to perform any software engineering task. If we recall the introductory example about NetBSD, in that situation, the reflexion model showed that the developer forgot to include a relation going from `FileSystem` to `Pager`. The dashed arc between `FileSystem` and `Pager` however is not enough to understand the nature of the relation. Indeed, as it is shown in Figure 3.2, in most of the cases the developer must also investigate the source model interactions mapped to particular arcs in the reflexion model. The inspection of the reflexion model can reveal either missing interactions in the high-level model or deficiencies in the map. For instance, an engineer could have specified a relation that links some source model entities with the wrong high-level model entity or in other cases the map may not be specific enough in capturing the user's intent. In similar situations the user will need to refine either the source model, the high-level model, or the map. To summarize, we can say that the refining process is about addressing divergences and absences. In general, in each refinement step the user computes and investigates successive reflexion models until he is completely satisfied. For instance, in the Excel example of Section 3.2 the Microsoft engineer faced many situations where functions were contained in files that no longer represented the module the file was supposed to represent. As a result of severals iteration, the initial map 170 entries transformed into a map with more than 1000 entries.

3.4 Problems and Limitations

The approach illustrated in Section 3.3 is limited by three majors problems:

1. Generation of the source model
2. Impossibility to understand the coverage of the high level model with respect to the real underlying system
3. Difficulty to compare results

In the next sections we analyze each problem in detail.

3.4.1 Source Model Generation

The source model, which is at the base of the reflexion model approach, is generated by an extraction tool. This tool allows the developer to extract structural information from the artifacts of the system. Source models can be generated in two different ways: with a static analysis of the system's source or by collecting information during the system's execution. During the experiment conducted by Murphy et al., the developer decided to extract the calls relation between the various NetBSD functions using xrefdb [MNS01]. The output generated by xrefdb could not be used straight away by the reflexion model tools and thus the developer wrote an awk script to convert it in a readable format. The model generated therefore depends primarily on two factors: the tool that extracts the information from the code and a component that translate the results in a suitable format for the reflexion model tools. This way of creating the source model is not optimal. First, the way of extracting the information from the source code is not standardized and thus the results might be difficult to compare. Additionally, the output needs first to be converted and this, despite being time consuming, opens the door to

possible errors. Last but not least, the way the source model is generated will have a great influence on the mapping step. Indeed, the user is completely free to decide what to include and what not. Therefore, the "mapping capabilities" of the modeler will be strongly entangled with the richness and expressiveness of the source model created by the tool chosen for the extraction.

3.4.2 Coverage

When a developer constructs a map as shown in Section 3.3.3, he creates a relation between the high level model and the concrete artifacts of the system. Murphy et al. defined the mapping such that it enables engineers to describe groups of source model entities using a hierarchical structural information and regular expressions [MNS01]. This also allowed the engineers to specify mappings where the number of entities involved was large. Despite being easy to use and scalable, this approach does not give any information about the portion of the system that is actually covered. In other words, if one simply looks at a map produced, it is hard to tell which part of the source model are being considered by the high level model and which are not. To better understand this problem consider the following example: imagine a developer wants to map his object-oriented system to the Model-View-Controller pattern. From a physical point of view, the system under analysis is composed by four distinct groups of files:

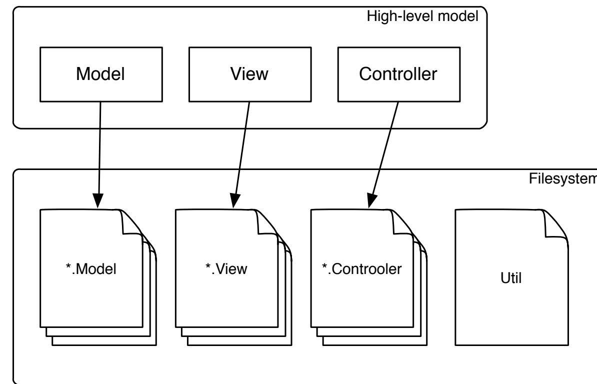


Figure 3.4. Mapping between high-level model and source files.

Group 1: classes whose name starts with `Model`
Group 2: classes whose name starts with `Controller`
Group 3: classes whose name starts with `View`
Group 4: single utility class named `Util`
The developer, as a first step, starts by defining three high-level model entities: the model, the view and the controller. After this, he starts the process of building a map. Before doing that, he quickly scans the physical structure of the source files in the file system by opening some folders and notices that there basically three kinds of classes:
Group 1: classes whose name starts with `Model`
Group 2: classes whose name starts with `Controller`
Group 3: classes whose name starts with `View`
Unfortunately, during his "random" scan the developer may miss a remote folder containing a class named `Util`. As a consequence, he may produce a map (shown in Listing 3.2) that contains three entries: one called `Model` that maps the classes whose name starts with Model, one called `Controller` that maps the classes whose name starts with Controller and one called `View` that maps the classes whose name starts with View.

Listing 3.2. Incomplete map

```
1 [file=Model.* mapTo=Model ]
2 [file=View.*  mapTo=View ]
3 [file=Controller.*  mapTo=Controller ]
```

At this point, the developer is confident of having an high level model that perfectly maps the system but the truth is that something is left, the `Util` class. The previous example shows that with this approach is very easy to leave out part of the software system without even noticing since there is no metric that can tell how much the current high-level model is covering the underneath source-model.

3.4.3 Result Comparison

The approach provided by Murphy et al. is made up by five different steps [MNS01]. In the first step, the user is required to extract from the software system under analysis a source model. The way in which the source model will be extracted is completely up to the user. For instance, in the NetBSD example the source model is extracted using the *xrefdb* tool while in the Microsoft Excel case the developer used an internal tool. Unfortunately, once the source model is generated there is no way of retrieving automatically which tool has been used for the generation. This is undesirable because it might lead to results that cannot be reproduced (e.g. only a Microsoft employee have access to the extractor used in the Excel example described in Section 3.2) or are impossible to compare since they are produced using different methodologies.

3.5 Summing Up

In this chapter we explored the concept of Reflexion Models presented by Murphy et al. [MNS01]. After providing an initial overview of the approach, as well as a basic terminology, we introduced an example in order to allow readers to understand the basic functioning of this approach. In addition to that, we illustrated the motivation behind the reflexion models and then we explained its building blocks. We concluded this chapter by enumerating the major shortcomings of this technique. The next chapter presents our main contribution, the Visual Reflexion Model approach, which represents an evolution of the original Reflexion Model technique.

Chapter 4

Visual Reflexion Models

In this chapter we present the concept of a Visual Reflexion Models which aims to overcome some of the limitations of the classical reflexion model approach we discussed in Chapter 3. We start with Section 4.1 that introduces a new technique for the representation of the source model. Specifically, we use a FAMIX meta-model that allows to represent the static structure of object-oriented software systems. Next, in Section 4.2 we illustrate the main building blocks of our approach that allows both to specify the high-level model and compute the resultant reflexion model. In Section 4.3 we move the focus to our approach that aims at building, analyzing, and refining reflexion models from a visual perspective. In that context, we illustrate the components of the user interface of our tool as well as providing a rationale for our choices. We end this Chapter with Section A where we explain from a technical point of view the architecture of our system by describing its components and their interactions.

Consider the architecture of the original reflexion model approach. Our contributions focus on the following components (Figure 4.1): (1-3) visual component to help developers define high-level models and mappings with the source model. (2) standardized source model (FAMIX). (4) visualization of the reflexion model. (5) tools for the inspection of the reflexion model.

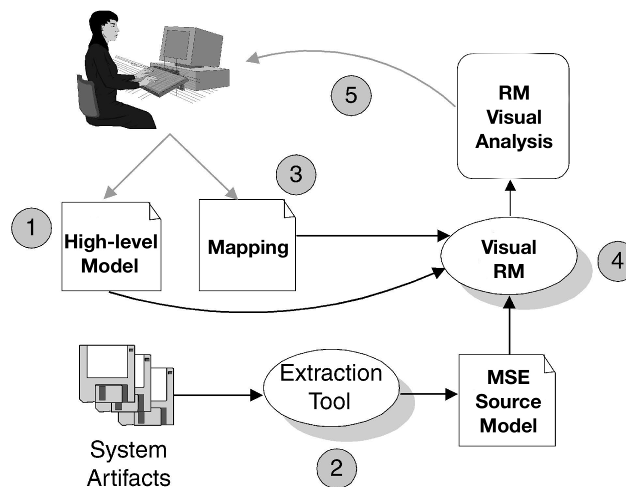


Figure 4.1. Visual Reflexion Model contributions

4.1 Source Model

The source model, as explained in Section 3.3, is at the heart of the Reflexion Model approach and can be imagined as the representation of the system being analyzed. One of the distinctive features of our approach is the source model and the way in which it is generated. More specifically, we decided that having the developer choose a tool to collect the information and later ask him to write a converter was not ideal. Therefore we choose to change the approach and use a tool that could do both the analysis and the production of an output that could be parsed in a second moment. This was possible thanks to a tool called VerveineJ, that allows to analyze a system, create a meta model of it, and export the results as an MSE file.

4.1.1 MSE

The MSE format, which is part of a meta-modeling framework called Fame[1], allows the specification of models. Analogously to XML, MSE is generic and allows the specification of any kind of data, regardless of the meta-model. Furthermore, MSE is simple, compact, readable and extensible. Compared to XML, the main difference is the absence of verbose tags in favor of parentheses to denote the beginning and ending of an element. The following snippet (Listing 4.1) defines 4 entities: 1 Namespace, 1 Packages, 1 Class and 1 Method. Each entity has an associated unique identifier (e.g., (id: 1)) and a set of properties. We can make a distinction between the kind of properties available: they can be either primitive, like (name `testNamespace`), or they can reference another entity, like in the case of (container (ref: 1)) which tells that the container property of ClassA is a pointer to the instance called `testNamespace`.

Listing 4.1. MSE file example

```
1  (FAMIX.Namespace
2    (id: 1)
3    (name 'testNamespace')
4  )
5  (FAMIX.Package
6    (id: 201)
7    (name 'aPackage')
8  )
9  (FAMIX.Class
10   (id: 2)
11   (name 'ClassA')
12   (container (ref: 1))
13   (parentPackage (ref: 201))
14 )
15 (FAMIX.Method
16   (name 'methodA1')
17   (signature 'methodA1()')
18   (parentType (ref: 2))
19   (LOC 2)
20 )
```

[1] http://www.moosetechnology.org/tools/fame

4.1.2 FAMIX

FAMIX is a family of meta models. The principal building block consists of a language agnostic meta-model that can represent uniformly both object-oriented and procedural languages. The core types entities are Namespace, Package, Class, Method, Attribute, and the relationships between them, namely Inheritance, Access and Invocation.

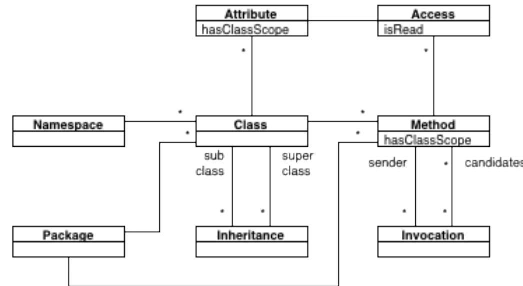


Figure 4.2. FAMIX core essential entities.

The main goal of a FAMIX model is a language independent representation of programs. Even if it is possible to represent many languages that share a common subset of features, it is important to realize that even if the meta model is language independent, the analyses that are built on top of it may have to be, as in our case, language specific.

4.1.3 VerveineJ

VerveineJ[2] is a tool created by Nicolas Anquetil that allows to analyze Java projects and export them into the MSE format. We also tried a similar tool, Infusion[3], but after some experiments we decided to drop it because the MSE produced was not as complete as the one produced by VerveineJ. Specifically, Infusion did not include any reference to the entities that were not explicitly mentioned in the source code. For example, in the MSE file generated with Infusion, we were not able to find any of the Java API's classes.

4.2 Backend

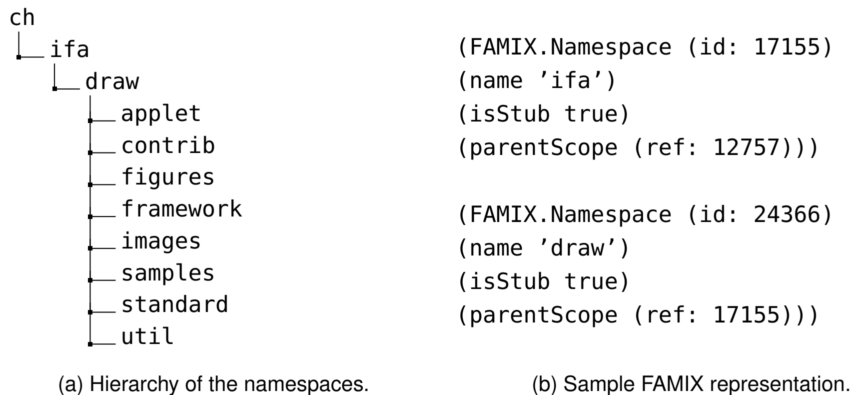
In the previous section we explained how we generate the source model from the source files of a project. The output generated by the tool, although containing all the information, is a simple text file and thus we cannot use it as it is. To exploit its contents we need to be able to query the FAMIX model and get insights about the system's composition. To do so, the MSE file is parsed and then all the relevant information about the system are reconstructed in a way that they can be accessed by means of a Java API. In the next subsections we explain the core components of our API, namely the source model entities, the mapping between source model entities, the high level entities and the relations between high level model entities.

[2] http://www.moosetechnology.org/tools/verveinej

[3] http://www.intooitus.com/products/infusion

4.2.1 Source Model Entities

In our system, source model entities represent the system under analysis in terms of namespaces, classes, methods and calls. Every entity has a name, a unique identifier and a list of contained model entities. Each entity, despite the basic attributes, has some unique characteristics. For example, the namespace entity contains the parent scope, if any, and a flag indicating if it is something used in the source code but its source is not available. If we consider the hierarchy shown in Figure 4.3a, our system will create an object for the `ifa` namespace that is going to have the `ch` namespace set as parent and the `draw` namespace in the list of contained models.



(a) Hierarchy of the namespaces.

(b) Sample FAMIX representation.

Figure 4.3. Namespaces of JHotDraw.

For the class entities we store information about the super-class, the eventual interfaces that are implemented and the namespace where it is contained. Last but not least for the method entities we save the signature, the return type, the container class, the methods called and the methods from which the method entity is called.

4.2.2 Entity Mapping

To consider a system from an higher level of abstraction, we need some way to identify and group source code entities that belongs together. Entities can be grouped either by using their textual attributes or by considering other structural features. In general, we can say that the application of a query (usually a regular expression) to one of the available types of mappings produces a group of source model entities.

$$\text{group of entities} = \text{mapping(query)}$$

In our approach we define seven different mappings. Specifically, we have four mappings that allow to group source-model entities and other three dedicated to create new mappings by composing existing ones. We now give a description for each one of them:

Name Mapping finds all the entities, whose name matches a given regular expression. For example, if we perform a query like `(.*)Figure` the mapping generated will contain all the namespaces, classes and methods whose name ends with `Figure` (e.g. `AbstractFigure`, `TriangleFigure`, `Figure`, ...).

Namespace Mapping finds the namespaces, if any, whose name matches the given regular expression. All the entities contained in the matched namespaces will be added to the mapping. In other words this mapping is very similar to the Name mapping except for the fact that it will not match neither classes nor figures. For example, if we perform a query like `(.*)Figure` the result will be an empty mapping since there are no namespaces in JHotDraw whose name ends in Figure. However if we do a query like `java(.*)` we will match two namespaces, namely: `java` and `javax`.

Import Package Mapping finds all the entities, in this case classes, that import a package specified by the user and adds them to the mapping.

Implement/Extends Mapping finds all the entities, in this case classes, that extend or implement an entity specified by the user and adds them to the mapping.

All the mappings described above can be composed in different ways to derive new mappings. The composition methods available are the union, the intersection and the complement.

Union Mapping the union of a collection of sets, in our case mappings, is the set of all distinct elements in the collection. More precisely, we have the union of a set of mapping `M` that in symbols can be written as $x \in \bigcup \mathbf{M} \iff \exists A \in \mathbf{M} \mid x \in A$ where M is a set whose elements are themselves sets.

Intersection Mapping the intersection of two mappings A and B is the set that contains all entities of A that also belong to B and vice versa. In our case intersections can be an arbitrary number of mappings. In symbols: $\left(x \in \bigcap \mathbf{M}\right) \Leftrightarrow (\forall A \in \mathbf{M} \mid x \in A)$ where M is a nonempty set whose elements are themselves sets.

Complement Mapping the complement, in this case relative, of A with respect to a mapping B is the set of entities that appears in B but not in A. In symbols: $B \setminus A = \{x \in B \mid x \notin A\}$

For example, we can create three mappings: the first containing all the entities whose name ends with `Tool`, the second with all the entities in the `draw` namespace and a last mapping with the entities that imports `java.awt.*`. Now we can make the union of the first two mappings and then compute the complement with the third. In that situation we have created a mapping that contains all the entities whose name ends with `Tool` that are contained in the `draw` namespace and do not import `java.awt.*`.

4.2.3 Mapping Relations

In a similar way as we explained in Section 3.3.3, we now have to define the mappings between the source model entities and the entities in the high-level model. We allow the users to define various kind of relations between mappings:

Call Relation is a relation from mapping A to mapping B meaning that there is at least one entity listed in A that invoke a function listed in one of the entities contained in B. For example suppose that we have a mapping called `Figure={Class:Figure,Class:TriangleFigure,...}` and a mapping called `Tool={Class:Tool,Class:ButtonTool, ...}`. There is a call relation between `Figure` and `Tool` if there exists a method in `Figure` (e.g. `Figure.draw()`) that calls a function in `Tool` (e.g. `Tool.render()`).

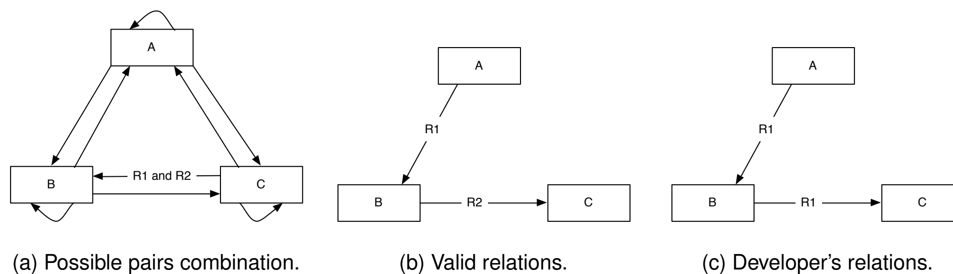
Containment Relation is a relation from mapping A to mapping B meaning that there is at least one entity listed in A that is physically contained in one of the entities contained in B. For example if the class `Figure` in A is contained in the `draw` namespace listed in B, we say that A is contained in B.

Superclass Relation is a relation from mapping A to mapping B meaning that there is at least one entity listed in A that is a subclass of at least one entity listed in B. For example suppose we have a mapping called `Abstract={Class:AbstractFigure,Class:AbstractTool,...}` and a mapping called `Concrete={Class:Tool,Class:Figure, ...}`. There is a superclass relation from `Abstract` to `Concrete` if at least on of the class listed in `Abstract` implements one or more class listed in `Concrete`.

Import Relation is a relation from mapping A to mapping B meaning that there is at least one entity listed in A that is stating among its import one of the entities in B. For example assume that we have a mapping called `Figure={Class:Figure,Class:TriangleFigure,...}` and a mapping called `View={Class:JPanel,Class:JFrame, ...}`. There is an import relation from `Figure` to `View` if at least on of the classes listed in `Figure` has an import referencing one or more classes in `View` (e.g. `import java.awt.JPanel` or `import java.awt.*`).

4.2.4 Reflexion Model computation

When the mappings are created and the relations among them are defined, it is possible to compute the reflexion model. The output of the computation will contain three distinct sets of edges: convergent, divergent and absent. In the first case we have the edges that represent the relations that were expected by the developer. In the second the edges listed are those that were not expected by the developer. In the last case the edges indicate interactions that were expected but not found. In order to compute the reflexion model, the system calculates the cartesian product between all the possible combinations of mappings and the types of relations available.



(a) Possible pairs combination. (b) Valid relations. (c) Developer's relations.

Figure 4.4. Reflexion model's computation.

To make this more clear we can consider the following example. Imagine we have three mappings A, B, C and two kinds of available relations R1 and R2. Given this input the developer specifies two relations (A,B,R1) and (B,C,R1). When the developer asks for the reflexion model, the system computes all the possible pair of combinations and checks their validity. More precisely the system will check the validity of all the eighteen triplets: from (A,A,R1) until (C,C,R2) as shown in Figure 4.4a. For the sake of this example, we assume that the valid triples found by the system are (A,B,R1) and (B,C,R2). Finally the system will compare the valid triples he computed against the triples provided

by the developer. As a result of this comparison the system will produce the reflexion model shown in Figure 4.5.

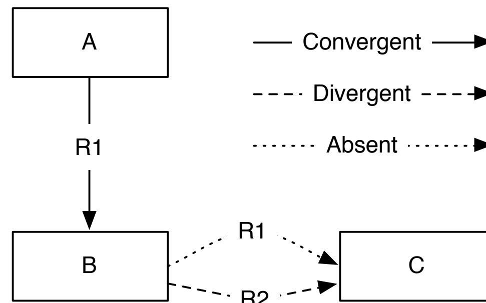


Figure 4.5. Reflexion model result.

4.3 Visualization

This section explains the most important feature of our work, the one that mostly distinguish our approach from the one explained by Murphy et al. [MNS01]. As we explained in Section 3.4, the original approach forces the user to create high level entities using a text based approach, limiting his creativeness and expressiveness. Additionally the original approach, although is made up of five clear steps, does not follow a clear flow since it is made up of several different and "disconnected" tools. One of our goal is to overcome this limitations by providing a visual interface that guides the user throughout the whole process: from the mapping phase to the inspection and refinement of the high level model. The starting point of our process is shown in Figure 4.6. At this stage we show to the user an interface composed by four macro-steps necessary to create the reflexion model. Namely:

1. **Upload**: upload of the project that the developer wants to analyze.
2. **Map**: definition of the high level model entities and mapping to the source model entities.
3. **Connect**: creation of the relations between high level model entities.
4. **Analyze**: generation of the reflexion model of the system.

Figure 4.6. Initial screen of our tool showing the four macro steps.

In the following sub-sections we show the details of all the visual interfaces described previously.

4.3.1 Mapping

In the mapping phase the user needs to define the high level model entities and their mapping with respect to the source model entities. As it can be seen in Figure 4.7, the interface for the mapping phase is composed by three main areas: a treemap, a sidebar and a top menu.

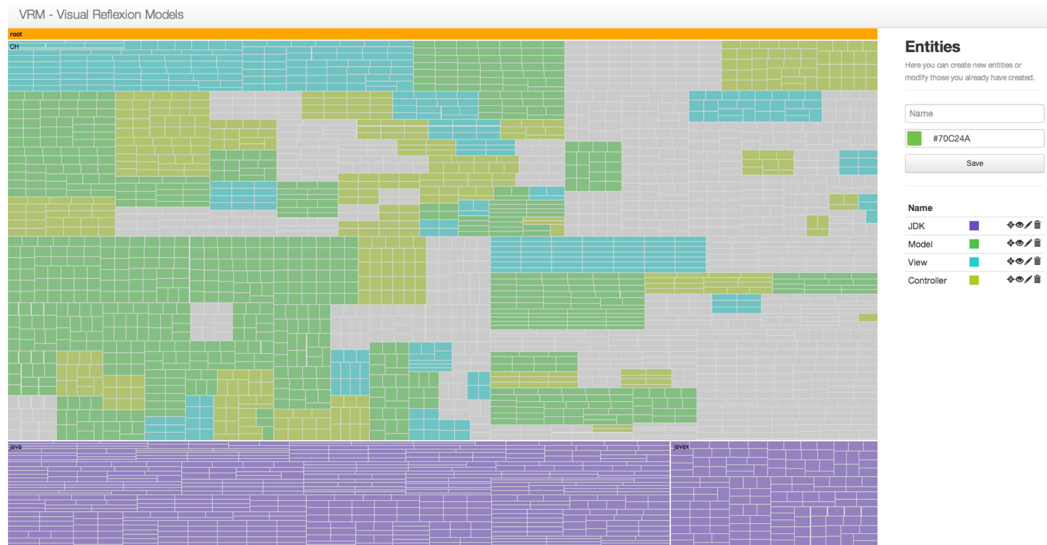


Figure 4.7. Definition of the MVC pattern on JHotDraw.

Treemap

Small structures can be easily represented by graphs. In the case of visualizing large information spaces the approach of Treemaps, proposed by Johnson and Shneiderman, has become an approved method [JS91]. A treemap is a method for displaying hierarchical (tree-structured) data by using nested rectangles. Each branch of the tree is given a rectangle, which is then sub-divided in smaller rectangles representing the various sub-branches. The area of the leaf node's will be proportional to a specified dimension on the data. The original treemap layout algorithm, called slice-and-dice, creates the map by recursively subdividing the the initial rectangle. The direction of each one-dimensional subdivision step alternates at each level: first horizontally, next vertically, again horizontally, and so on. This way of layouting the treemap considers only one dimension at each step and, as a result, it can produce a large number of elongated rectangles with a high aspect ratio between width and height. This kind of elongated rectangles, as it is shown in Figure 4.9a, are particularly difficult to select, compare and label. This issue is addressed by Bruls et al. with the introduction of Squarified Treemaps [BHVW00]. The main contribution of their work is an improved layout algorithm that, by working on both dimensions (hori-

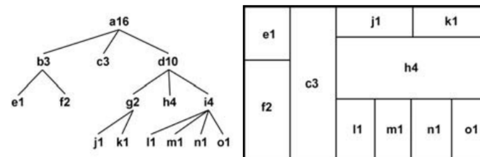
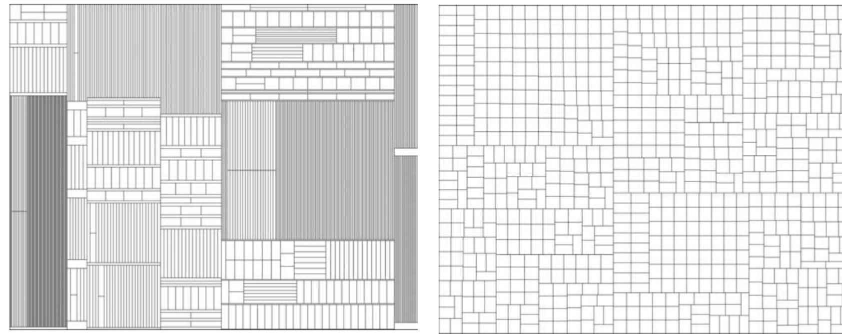


Figure 4.8. Tree diagram and corresponding Treemap.

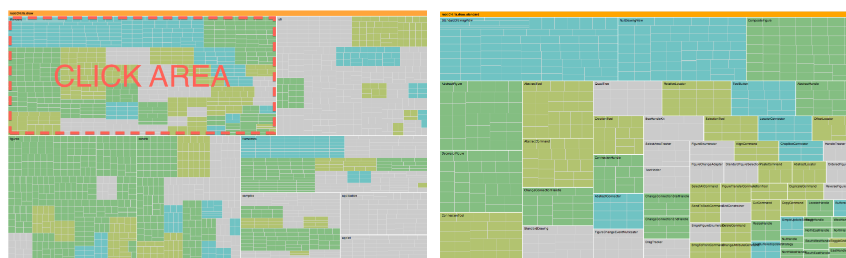
zontal and vertical), tries to approximate each sub-rectangle to the shape of a square, whereby the aspect ratio between the width and height of each rectangle tends to one. With this kind of algorithm a treemap acquire a cleaner, and thus more understandable layout. This can be seen in Figure 4.9 where we can easily understand the benefits of the squarified layout shown in Figure 4.9b.



(a) Slice-and-dice algorithm. (b) Squarify algorithm.

Figure 4.9. Layouting algorithm for treemaps.

Given the previous introduction, in this work we decided to use a squarified treemap to represent the source model. Even though we found a workaround for limiting the number of elongated rectangles it was still impossible to give a label to each one of them. All the information in the source model are equally important and thus they should be included in the treemap. Given this constraint, we decided to create a custom solution: an interactive squarified treemap. More precisely, in our implementation the treemap shows only the rectangle's labels of the entity's children at the current depth of the tree. When the user clicks on a section of the treemap the depth increases by one and the viewport moves focusing only on the part that was clicked by the user. Consider for example Figure 4.10a, at that moment the user is inspecting the `draw` namespace (third level of the hierarchy shown in Figure 4.11) and thus the rectangles labeled are only its immediate children (i.e. `standard`, `util`, `draw`, ...). After the user clicks on the portion of the map that has the label `standard`, the depth increases, the viewport is adjusted and the labels of the entities contained in `standard` appears.



(a) Before click. (b) After click.

Figure 4.10. Interactive treemap layout.

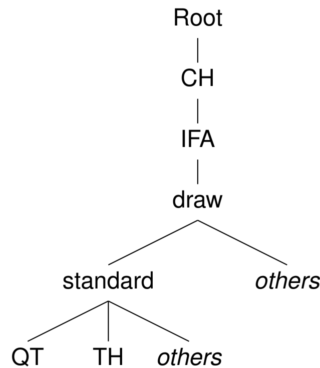


Figure 4.11. Partial Tree representation of JHotDraw source model

In order to allow the user to click on portions of the treemap to focus on particular entities, our interactive treemap is organized in layers as it is shown in Figure 4.13b. Specifically, in Figure 4.13b we have the `ifa` namespace as the top layer, under that it is placed the `draw` namespace and at the bottom we have all the various classes contained in `draw`. Our interactive treemap provides also a header, shown in Figure 4.12 that allows the user to see the name of the parent entity and going back to the previous level of the system's hierarchy if it is clicked.

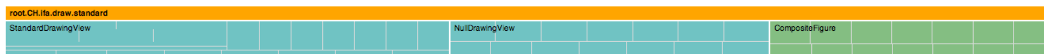
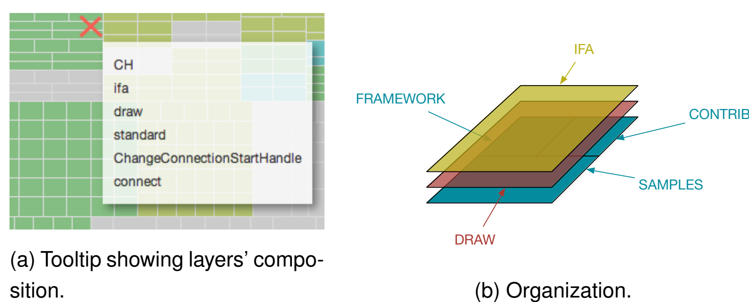


Figure 4.12. Header of the treemap.

As explained previously, labels are discovered progressively as the user goes down into the hierarchy. In some cases the developer has not a clear idea of the system and navigating up and down the structure might become very tedious, especially for large and complex systems. To overcome this problem, we decided to facilitate the user by providing a tooltip (Figure 4.13a) that shows the composition of the layers at a given position in the treemap.



(a) Tooltip showing layers' composition.

(b) Organization.

Figure 4.13. Treemap Layers.

In addition to the navigating capabilities, our interactive treemap provides the user other functionalities:

Scope Coloring allows entities to be colored in different ways according to the task performed by the user. By default all the entities of a particular high level model have the same color. However, if the user wants to inspect an entity in more detail the various mappings belonging to it are given different colors so the user can better understand the composition. Consider for example an high level entity named `Model` that contains all the source model entities whose name ends with `Figure` and those whose name ends with `Handle`. On the default view (Figure 4.14a) they appear as a single block but when they are inspected in detail they appear as separate (Figure 4.14b).

(a) Default mode. (b) Inspection mode.

Figure 4.14. Entities coloring.

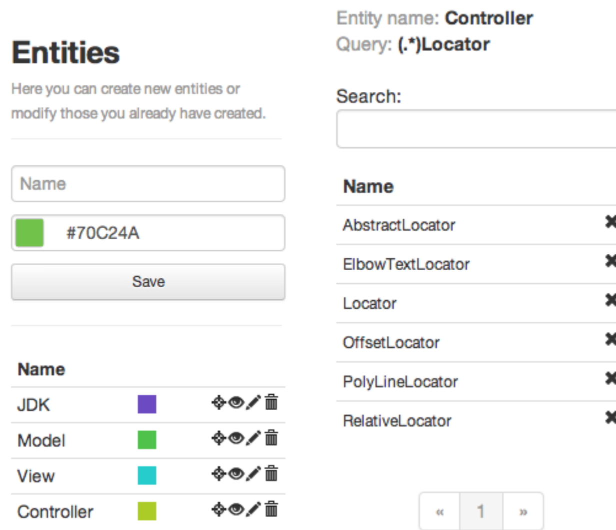
Elision allows the developer to hide something completely from the treemap and can be done both on an high level entity as well as the source model entities. This action can be handy in those situations when the developer wants to focus on what is still not mapped.

Focus is the opposite of elision and allows to focalize on an high level entity by eliding everything apart from the entity itself. This can be useful when the developer is dealing with many high level entities and wants to focalize only on a specific one discarding all the rest.

To summarize, our interactive squarified treemap allows the users to easily explore the system as well as composing the high level entities.

Sidebar

The sidebar adapts to the current task of the user. By default it enables the developer to perform various operations on the high level models. More precisely it allows to create, delete, focus and elide entities as we described in the previous section. Creating a new entity is as simple as selecting a name and a color. When the entities are created, or an existing one edited, the content of the sidebar changes by showing the user the mapping that is currently selected and the source model entities that it contains. Figure 4.15 shows on the left, the default sidebar. In that case we can see that there are four high level model entities created: `Model`, `View`, `Controller` and `JDK`. For each one of them it is shown the name, the color and the possible actions. On the right side, we have the sidebar as it looks like when the user wants to edit an entity. In this case we have a high level entity named `Controller`. Inside that entity we have a name based mapping (`(.*)Controller`) that maps every entity whose name ends with `Controller`. The remaining part of the sidebar contains the source model entities



(a) Creation (b) Editing

Figure 4.15. Different views of the sidebar.

contained in the specified mapping. For each one of them, the only action possible is to exclude it from the mapping.

Top Bar

The top bar allows the user to manipulate mappings. More precisely, it allows the inspection of the current mappings using a dialog box (Figure 4.16) and allows the user to create new ones. For example, in Figure 4.17 we can see that for the current entity the bar shows that we have saved three different mappings namely: `(.*)Tool`, `(.*)Command` and `(.*)Locator`. For each mapping that has been saved the user can edit it, delete it or elide it. The right side of the top bar is dedicated to the creation of mappings: in that area the user can selects the kind of mapping desired (among those we described in Section 4.2.2), insert the parameters necessary and select a color that will identify that particular group of source model entities.

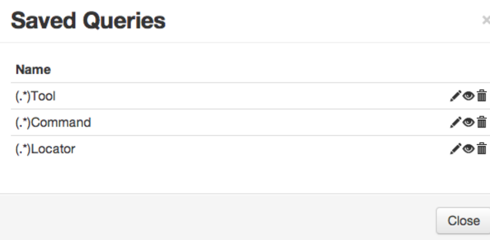


Figure 4.16. Dialog box showing saved mappings.

Figure 4.17. Top bar aspect when editing an high level entity.

4.3.2 Connection

When the user is satisfied with the high level entities he built in the previous step he needs to define what are the relations among them. To represent the various high level model entities we decided to use colored squares labeled with the name associated with that particular entity (Figure 4.18). In order to define an interaction between two entities the user needs to click first on the source and then on the target. When the user clicks on the source the entity that has been selected starts to blink. The interaction will be defined only when the target will be clicked. In Figure 4.19 we can see this process: first the user clicks on the `Model` and then in order to define a relation clicks on the `Controller`.

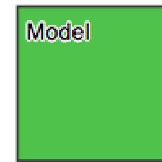
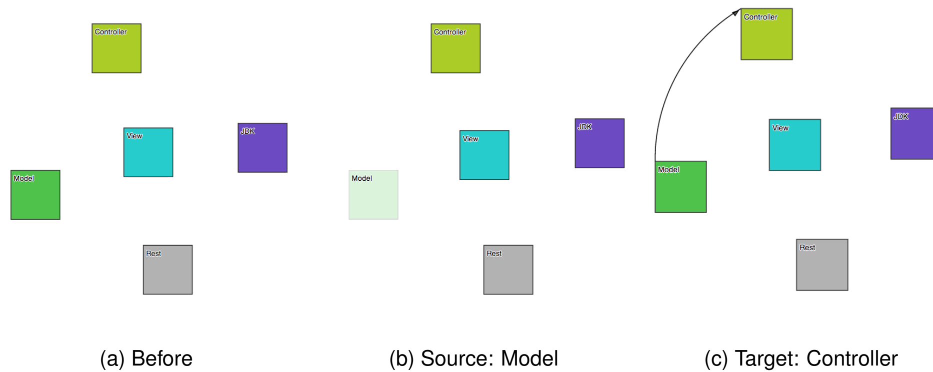


Figure 4.18. Entity representation during linking phase.

Despite showing all the entities created during the mapping phase there is an extra gray square named `Rest`. This square is different from the others and represent a mapping including all the source models that were not included inside any of the high level model entities.



(a) Before (b) Source: Model (c) Target: Controller

Figure 4.19. Linking two entities.

4.3.3 Analysis

This part is the final step of our approach. We decided to give the users the ability to analyze the interactions computed in the reflexion model at different levels of granularity. Specifically, there are three different types of analysis: high level models relations, mapping relations and source model relations. We will now see in detail each one of them.

High level model

Given the stated high level entities and the relations among them, the developer asks the system to compute the reflexion model. When the system is done with its computation it will add to the graph created in the previous step the various edges. More precisely, as we already explained in Section 5.2.3, it will add the convergent, divergent and absent edges. For example, in Figure 4.20 and in Table 4.1 we can see that the reflexion model produced contains all those kinds of edges.

Table 4.1. Edges in the Reflexion Model

Source	**Target**	**Type**
Model	Controller	Convergent
View	Controller	Convergent
View	View	Absent
View	Model	Absent
View	Rest	Absent
View	JDK	Absent
Controller	View	Absent
Controller	Controller	Absent
Controller	Model	Absent
Controller	Rest	Absent
Controller	JDK	Absent
Model	View	Absent
Model	Model	Absent
Model	Rest	Absent
Model	JDK	Absent
Rest	View	Absent
Rest	Model	Absent
Rest	Rest	Absent
Rest	JDK	Absent
JDK	Rest	Divergent

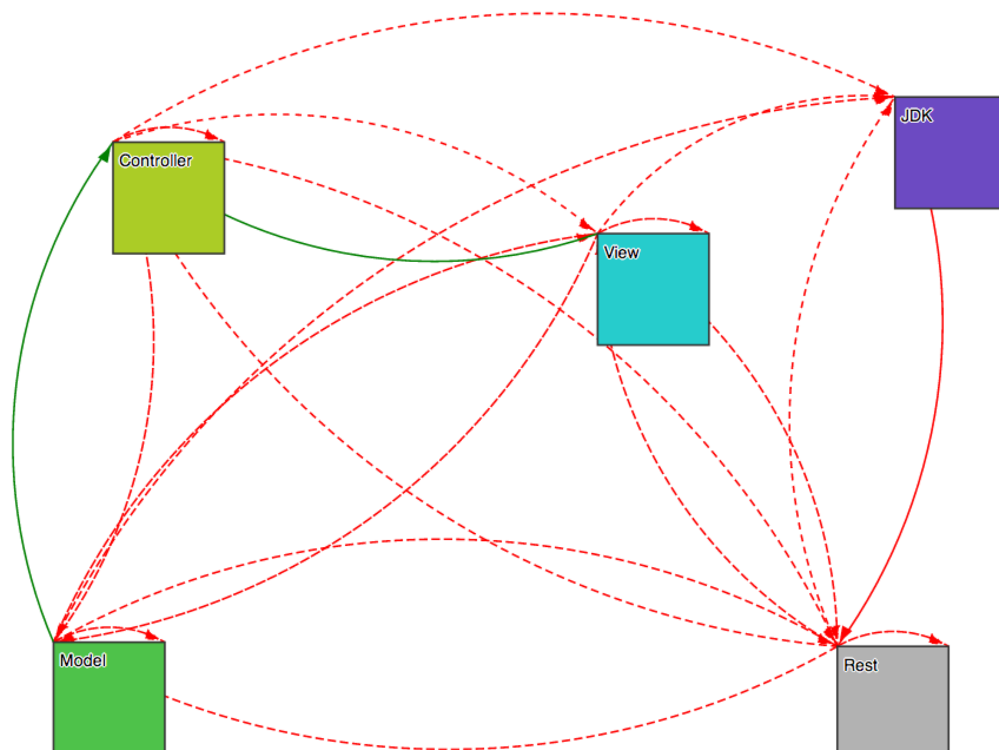


Figure 4.20. Reflexion Model.

Mapping

Inspecting the links among high level entities might be not enough to understand what is the real nature of a specific relation. In fact, there are situations where a link is present only because there is a relation between a mapping in the source and a mapping in the target. For example, consider that we have a high level model entity called `Model` that is the source of our relation. In that entity there are a lot of mappings, one of those is matching all the classes whose name ends in `Figure`. On the target side, we have an high level model entity called `Controller` that among all of its mappings has one that maps all the entities whose name end in `Locator`. If there's some kind of relation among those two mappings we create a relation between the two high level model entities, no matter what the other mappings are doing. Consider for example Figure 4.21: in that case we can see that the the relation between the `Model` and the `Controller` is caused by the relations between the mappings `Handle`,`Connection` and `Figure` with the mapping `Locator`. From a purely visual point of view we use again the metaphor of the square to represent the entities, in this case mappings. To enforce the idea of mappings being part of a group, the color of the squares is taken from the high level model entity in which they are contained. Additionally those two groups are graphically separated as well as contained in two separate convex hulls.

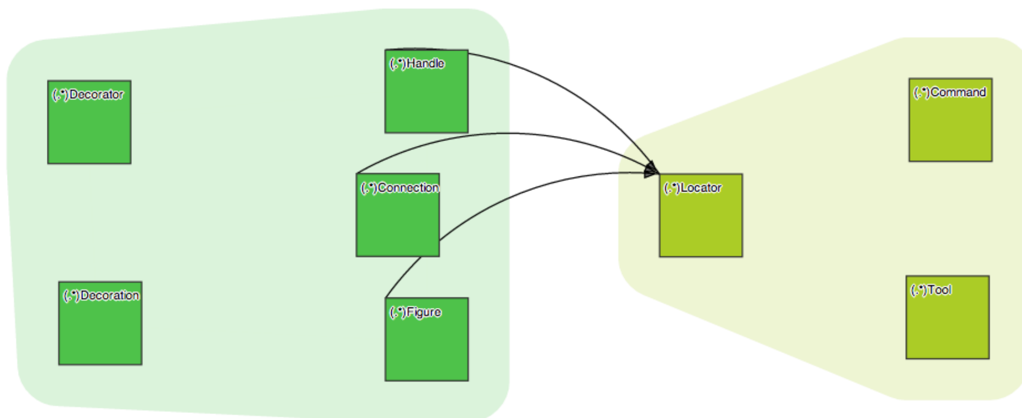
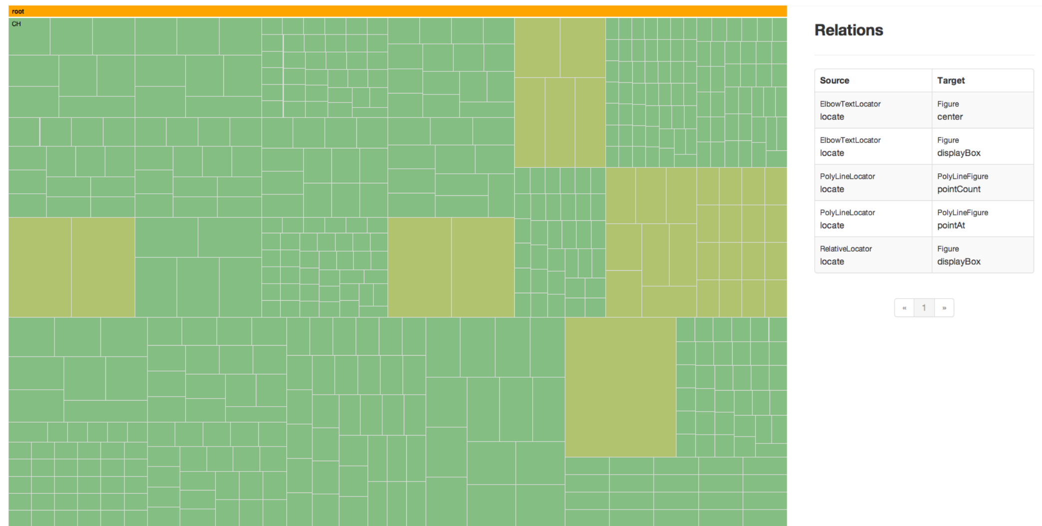


Figure 4.21. Relations between mappings in different high level model entities.

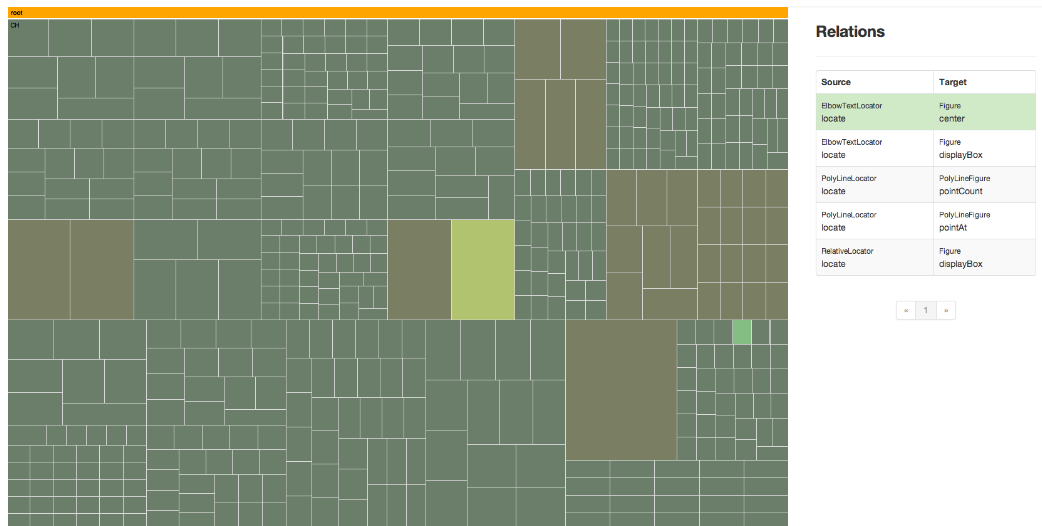
Source Model

As it happens for relations between high level model entities, for similar reasons it is non-trivial to understand what is the nature of the relations between two mappings. To help developers acquiring an even better understanding of the system being analyzed, we allow them to inspect also the relations down to the level of source model entities. The number of entities in a mapping can be considerably high. For example, if we have a mapping that matches everything contained in the `java` namespace we would have more than 4000 entities. For this reason it was not possible to use the same visual approach we adopted for showing the relations among mappings and thus we opted again for the interactive squarified treemap we explained before. In this case the treemap contains just the two mappings under analysis and the entities are filled with the same color used by the high level model entity in which they are contained. On a sidebar next to the treemap we list all the relations in a tabular form. More precisely in the first column we list all the sources and in the second and last column we

list all the targets. To facilitate the localization of the two entities involved in the mapping we allow the developer to highlight them by clicking on the row that represent the relation he wants to analyze.



(a) Default



(b) Relation highlighted

Figure 4.22. Inspection of a relation between two mappings.

4.4 Summing Up

Chapter 4 presented the concept of Visual Reflexion Model by explaining the main building blocks of our approach and how it allows to overcome the limitations of the original approach presented in Chapter 3. Specifically, we started by introducing our visual approach for the creation and refinement of reflexion models. Next, we presented the idea of an interactive squarified treemap for the visualization of meta-models. Last but not least, we illustrated a web-based application that shows our approach in action. In Chapter 5 we discuss two case studies that aim to verify the effectiveness of our idea.

Chapter 5

Evaluation

In Chapter 4 we presented our approach that aims at overcoming the limitations of the reflexion model approach. We believe that our approach can aid software comprehension by helping the user reverse architecting a software system. In this chapter, we construct two case studies by applying our tool on different software systems to draw conclusions about our approach that will give us directions for further improvement. In the first case study (Section 5.1) we try to understand, with the help of our tool, the application of the Model-View-Controller pattern (Figure 5.2a) [GHJV94]. More specifically, we decided to analyze if the advertised design patterns in JHotDraw documentation are correctly applied. In the second case study (Section 5.2) we consider a possible general architecture of ArgoUML and we analyze it in order to find out possible discrepancies with the actual system.

5.1 JHotDraw

JHotDraw[1] is a Java framework to create drawing editors (Figure 5.1). It has been successfully employed in a wide a range of programs: from simple Microsoft Paint like editors to more complex programs that have rules about how their elements can be used and altered, like a UML diagramming tool. In general, JHotDraw provides support for creating geometric and user defined shapes, editing those shapes, animate them, and adding behavioral constraints in the editor. In the following sections we start by defining the Model-View-Controller architecture (Section 5.1.1) which we then use to define the high-level model entities (Section 5.1.2). After that, we specify the relations (Section 5.1.3) and then we compute a first reflexion model (Section 5.1.4). Lastly we analyze the reflexion model and we refine it (Section 5.1.5).

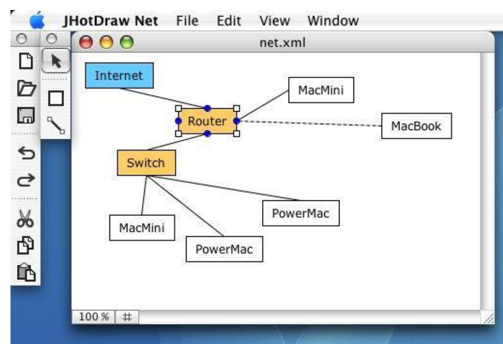
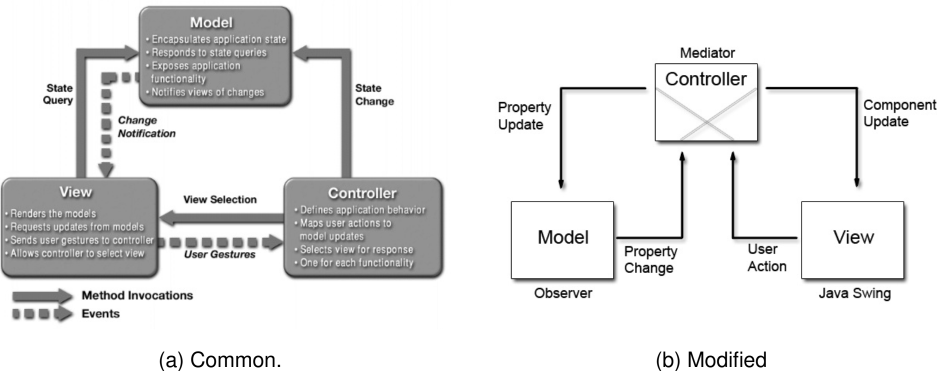


Figure 5.1. Sample application built with JHotDraw.

[1] http://www.jhotdraw.org/

5.1.1 Model View Controller

The MVC pattern is very common and it prescribes the role of classes to belong to three categories:



(a) Common.

(b) Modified

Figure 5.2. M-V-C pattern implementations

Model represents data and establishes how it can be accessed and updated.

View provides a visual representation of the model and accepts user's input. If the model data is changed, the view must update its presentation as needed. This can be achieved either using a push or a pull approach: in the first case the view registers itself with the model for change notifications while in the second case is the view that is responsible for calling the model when it needs to retrieve updated information.

Controller is responsible for the mediation between the view and the model; it translates the user's requests (i.e. actions performed on the view) into actions that the model will perform.

Those three groups have well defined interactions. Usually once the Model, View and Controller components are instantiated the following three actions occur:

1. In the push model scenario, the view registers itself as a listener into the model. As a result of this registration, the view will get a notification as soon as the model is changed. In this pattern the model is not aware neither of the view nor of the controller. Indeed, it simply act as a broadcaster to all the registered listeners.
2. The controller gets bounded to the view. In other words, any action the user performs the view will trigger a listener method in the controller class.
3. The controller is linked to the underlying model.

When the user interacts with the view, the following actions occur:

1. The view detects a GUI action (e.g. a button press) using a listener method that is registered so that it gets called when such an action occurs.
2. The view invokes the appropriate action on the controller.
3. If it is required the controller accesses the model, updating it in accordance with the user's action.
4. If the model is changed it sends a broadcast message to the interested listeners (e.g. the views).

The MVC pattern comes also in a different flavor where the controller is placed between the model and the view. This way of interpreting the pattern is shown in Figure 5.2b and it can also be found in the Apple Cocoa framework[2]. This particular design of the MVC pattern is different from the original

[2] https://developer.apple.com/technologies/mac/cocoa.html

approach because the notifications of state changes in the model are broadcasted to the view passing from the controller. In this sense, the controller acts as a mediator of the flow between model and view object in both directions. As in the traditional MVC pattern, view objects use the controller to translate the actions performed by the users into updates performed on the model. In contrast, changes in the model are not broadcasted directly but are first communicated to the application's controllers that will then be responsible of notifying the views.

5.1.2 High-level model entities

As it is pointed out in the official website[3], JHotDraw adheres to the MVC guidelines. Every application built using JHotDraw, in most of the cases, is the result of the interaction between:

a) Swing related classes that take care of the View management
b) A thin controller
c) Model classes written by the developer that depends on the type of software being written

Referring to Figure 5.3, the `Drawing` interface can be interpreted as the model given that represents the current state of the application, while the `DrawingView` interface represent the actual GUI containing the drawing. JHotDraw's graphical objects are manipulated with the help of `Tools`, which in that sense partially play the role of the controller.

Unfortunately, as [Kai01] points out, "JHotDraw is not MVC at its purest" because of the way events are notified between the models and the views.

With the help of our tool, we now investigate further the design conformance of JHotDraw with respect to the MVC pattern. After having uploaded the zipped project to our tool, the first thing we do is the creation of three high-level model entities:

- Model
- View
- Controller

After that, we need to define the various mappings. In other words, we need to decide which source code entities are associated with which high-level model entity. We now discuss each one of them in detail.

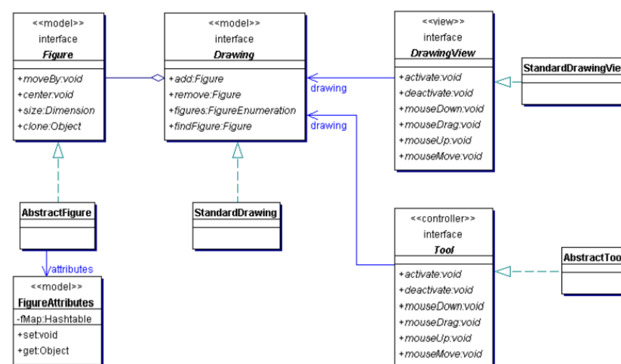


Figure 5.3. UML diagram of JHotDraw.

[3] `http://softarch.cis.strath.ac.uk/PLJHD/Patterns/JHDDomainOverview.html`

Model

From Figure 5.3 we can see that the five main source-model entities for the Model are `Figure`, `AbstractFigure`, `FigureAttributes`, `Drawing` and `StandardDrawing`. For each one of them we specify a mapping. From a visual point of view, this initial set of mappings covers only a small part of the whole source model. Specifically, in Figure 5.4 we can see the followings associations:

(1) AbstractFigure
(2) StandardDrawing
(3) FigureAttributes
(4) Figure
(5) Drawing

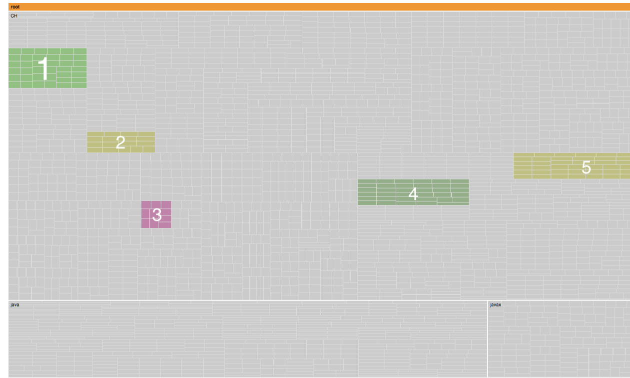


Figure 5.4. Coverage of the high-level entity Model.

View

In this case, if we consider again Figure 5.3, we can notice that the View contains only two source-model entities: `DrawingView` and `StandardDrawingView`. Also in this case we specified a mapping for each one of them. From a visual perspective (Figure 5.5) we have the following associations:

(1) StandardDrawingView
(2) DrawingView

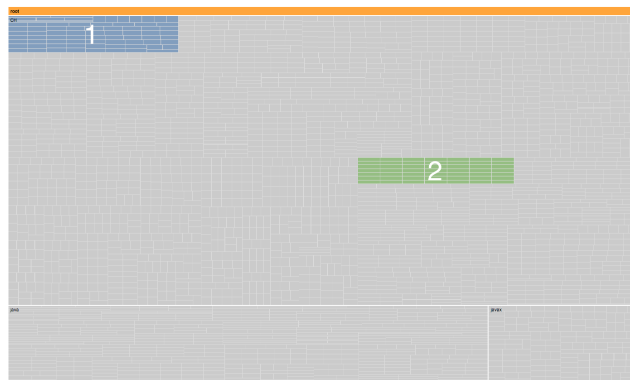


Figure 5.5. Coverage of the high-level entity View.

Controller

As for the previous cases, if we consider again Figure 5.3, we can notice that the Controller contains two source-model entities: `Tool` and `AbstractTool`. After having specified a mapping for each one of them, from a visual perspective (Figure 5.6) we have the following associations:

(1) AbstractTool
(2) Tool

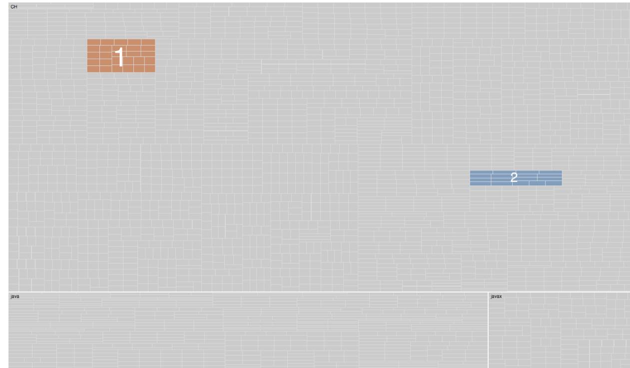
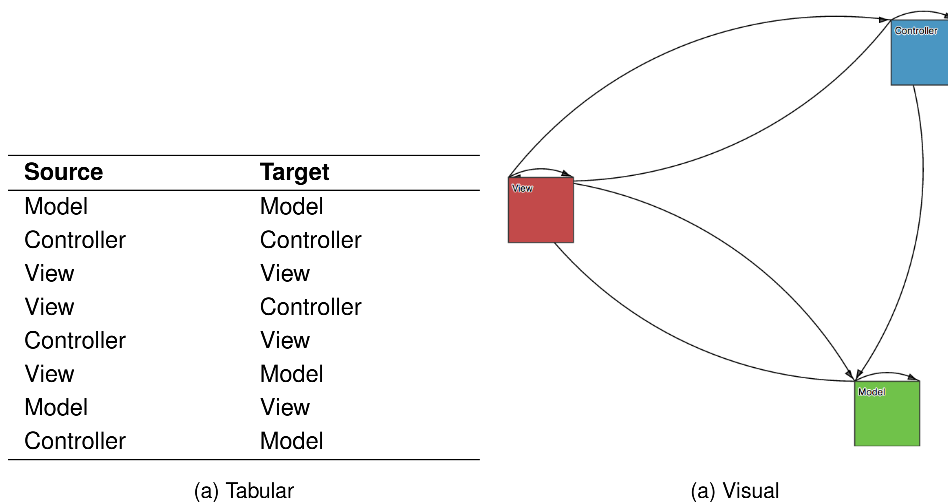


Figure 5.6. Coverage of the high-level entity Controller.

5.1.3 High-level model entities relations

At this point, we have all the high-level model entities in place. What we have to do now is to define how those entities relate to each other. To check the design conformance with respect to the MVC pattern, we define the various interactions as they are exposed in Figure 5.2a. Specifically, we defined 8 relations as we show in Table 5.7a as well as in Figure 5.8a.



Source	Target
Model	Model
Controller	Controller
View	View
View	Controller
Controller	View
View	Model
Model	View
Controller	Model

(a) Tabular

(a) Visual

Figure 5.8. Links between MVC entities in JHotDraw.

5.1.4 Reflexion Model computation

After we defined all the relations, we are finally ready for the last step: the computation of the reflexion model. In this case, we would expect that the reflexion model confirms the high-level model we created. Unfortunately this is not the case. Indeed, the reflexion model (Figure 5.9) shows that there are two relations that, despite being present in the high level model, are not present in the reflexion model. Specifically, those two relations are: the one that start from `Model` and goes to the `View` and the one that starting from the `Controller` goes to the `Model`. As we already anticipated, JHotDraw is not MVC at its purest and thus a similar result is tolerable. The reason of the missing connection between the `Model` and the `View` can be attributed to the fact that, in this case, the two entities interact using events rather that methods invocations. Unfortunately, this behavior is not captured inside the FAMIX model upon which we generated the source model. This lack of information is due to the fact that VerveineJ performs a static analysis on the source code and thus events that happen at runtime, such as registrations and notifications, are not captured. For the other missing connection, that goes from `Controller` to `Model` we can assume that it is due to an improper application of the MVC pattern.

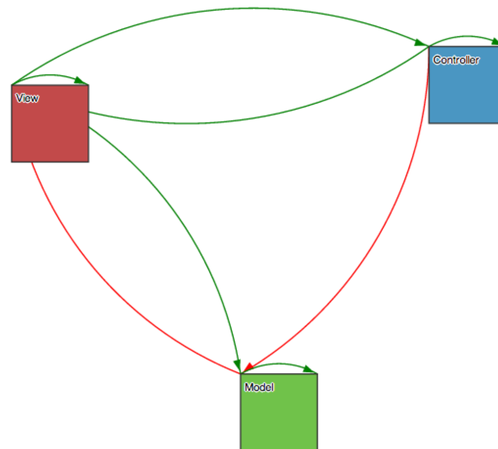
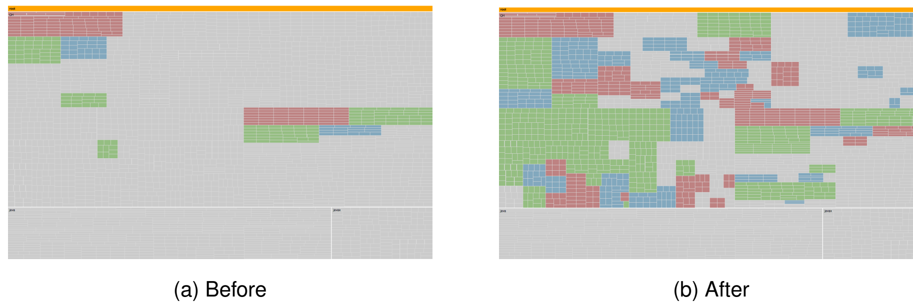


Figure 5.9. Reflexion Model of JHotDraw

Thanks to our tool, we can immediately see that the current high-level model only maps a small part of the whole system. In the next paragraphs, we gradually refine our initial high-level model in order to achieve a better coverage of the system.

5.1.5 Refinement

As as starting point of our refinement process we start by looking at the “rest” (i.e. the source model entities that does not belong to any mapping). A first inspection reveals that our high-level model does not consider any sub-classes and thus classes like `DiamondFigure`, which intuitively is a subclass of `Figure`, were not included inside any of the mappings.



(a) Before (b) After

Figure 5.10. Refinement of JHotDraw's high-level model.

After including all the relevant source-model entities in their respective high-level model entities we have a great improvement with respect to the coverage of the system. Figure 5.10 shows the system coverage before and after the refinement process. Most of what is left is either related directly to the Java SDK or in other cases they are not strictly related to the core of JHotDraw (e.g. utilities classes like `ReverseFigureEnumerator`).

The computation of a new reflexion model (Figure 5.11), based on the refinements of the high-level model we discussed previously, gives us an interesting result. More precisely, we can see that compared to the reflexion model shown in Figure 5.9 there are two major changes:

1. The relation going from `Controller` to `Model` and the relation going from `Model` to `View` are now convergent.
2. A new relation that goes from `Model` to `Controller` emerged from the reflexion model.

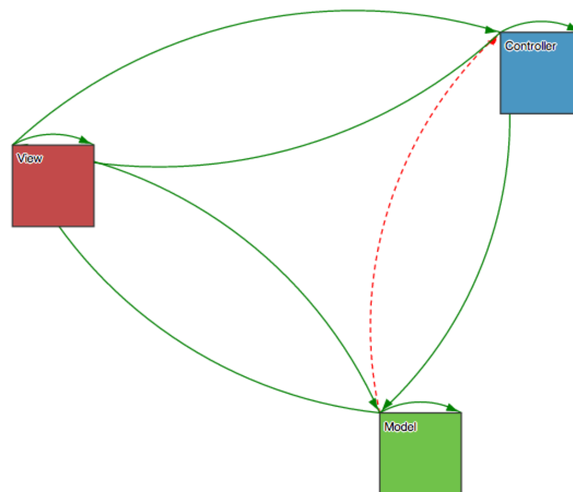


Figure 5.11. Reflexion Model of JHotDraw after the refinement

To understand if those changes reflects the reality, we will consider each one of them in detail.

From Model to Controller

This relation was not present in the first reflexion model and it is not included inside the high-level model, therefore we need to inspect it to understand why this happens. By inspecting the connections between the mappings of `Model` and `Controller` (Figure 5.12), we can see that the relation is caused by the connection of some of the mappings in the `Controller` with the mapping `(.*)Locator`.

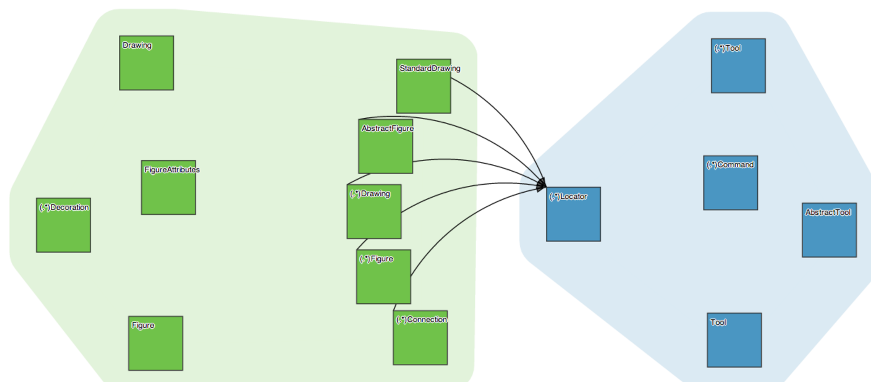


Figure 5.12. Inspection of the relations between Model and Controller

In JHotDraw Locators are used to locate a position on a figure. In general, they are employed to define where handles or connectors should be placed on a figure. We assumed that those kinds of source-model entities performed only computations without carrying any information about the current status of the system. However, the truth is that Locators should be considered as part of the `Model`. Consider for example the `RelativeLocator`: this class is used to represent positions relative to a figure boundary. To allow this functionality, the object stores information about the relative position with respect to another object. Given the previous motivation we move the `Locator` mapping to the `Model`.

From Controller to Model

This relation was specified in the high-level model but it was not present in the first reflexion model computed. In the refined reflexion model, however, this relation became convergent. The modified reflexion model agrees with the MVC pattern, shown in Figure 5.2a, where the Controller calls the Model. We now inspect the nature of this relation in order to confirm its “correctness”. Figure 5.13 shows that two entities are tightly connected. Specifically, we can see that both the `Tool` mapping as well as the `Command` mapping have lots of connections with the `Model` entity. In JHotDraw, the Tool-related classes are used to capture mouse events while the Command-related classes let toolkit objects make requests of unspecified application objects. By inspecting the relations among the various mappings, as it is shown in Figure 5.14, we gained enough confidence to confirm the goodness of our high-level model.

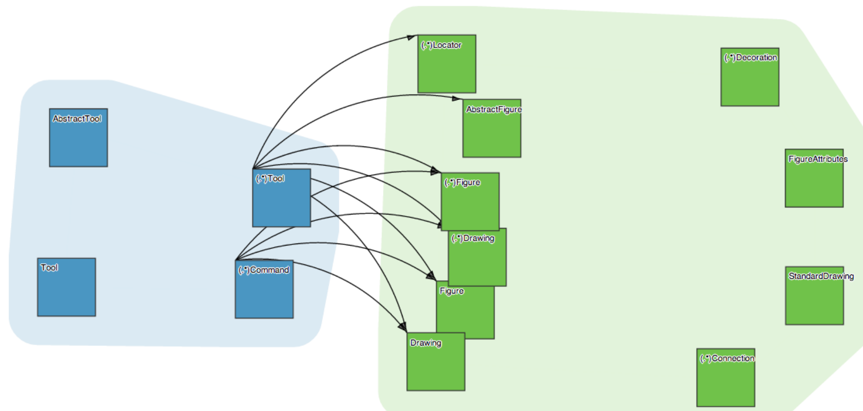


Figure 5.13. Inspection of the relations between `Controller` and `Model`

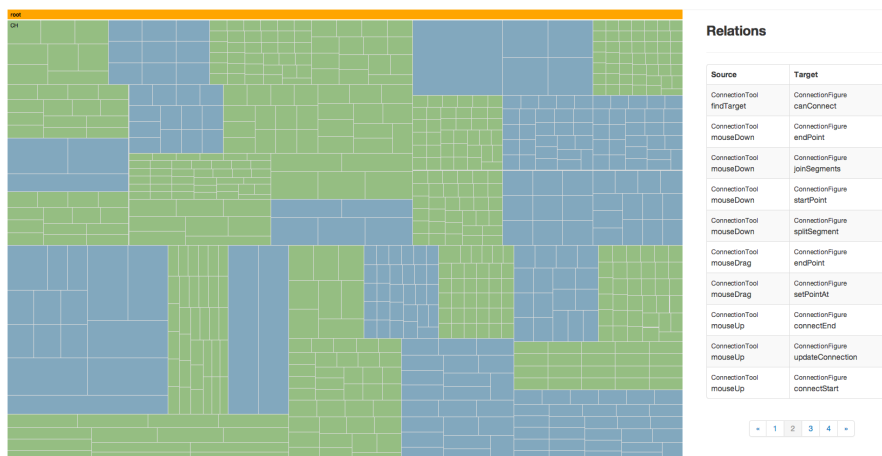


Figure 5.14. Inspection of the relations between `(.*)Tool` and `(.*)Figure`

From Model to View

In the ideal situation, as we explained previously in the introduction about the MVC pattern, the `Model` and the `View` should communicate by means of events. This kind of information, however, is available only at runtime and thus it is not contained in the source-model we are generating. Despite this, in the refined reflexion model shown in Figure 5.11, we have a convergent relation going from the `Model` to the `View`. By inspecting this relation we can understand if it is caused by a wrongly constructed high-level model or by a simple misuse of the MVC pattern. In Figure 5.15, we can see that all the relations either have as a target `(.*)Handle` or `(.*)Connector`. As a further step, we need to expand our investigation to understand what is the nature of the relations among the various mappings. To do so we we push our analysis to a deeper level by considering the relations among the various source-model entities. This new inspection (Figure 5.16) reveals that the single entities in the target mapping that are involved in the relation, are more likely to be part of the `Model` rather than be part of the `View`. Consequently, in this case, we can say that the presence of a convergent relation going

from the `Model` to the `View` is due to a wrongly constructed high-level model. This can be confirmed by reading the JHotDraw documentation where it is clear that both `Handles` as well as `Connectors` should be considered part of the `Model`.

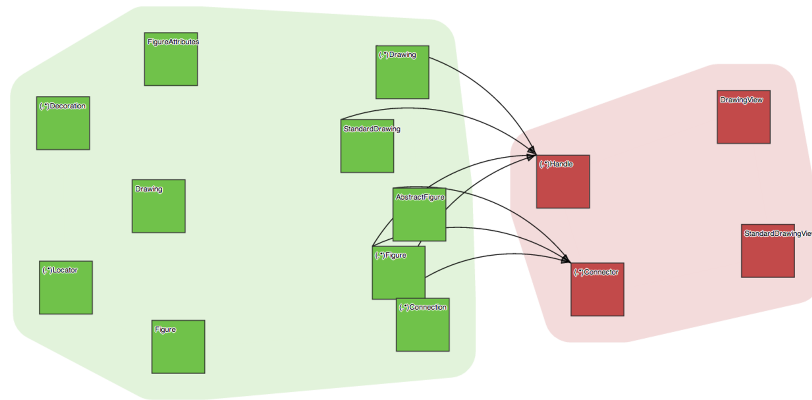


Figure 5.15. Inspection of the relations between `Model` and `View`

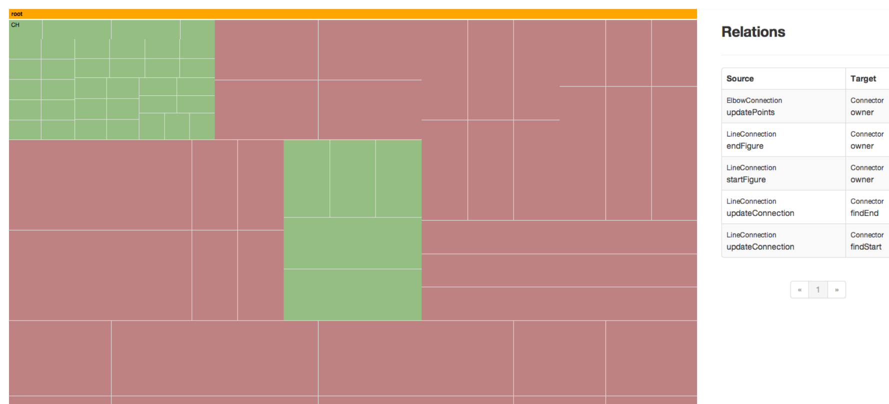
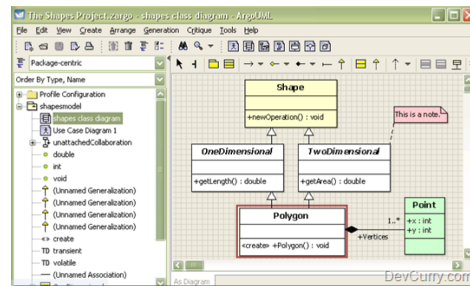


Figure 5.16. Inspection of the relations between `(.*)Figure` and `(.*)Connector`

Remarks on the *rest*

In the previous analysis we deliberately omitted the *rest* entity. This entity, as we explained in Section 4.3.2, represents all the source model entities that does not belong to any mapping. In our case, the *rest* entity contains every source model entity that is not part neither of the `Model` nor of the `Controller` nor of the `View`. We decided to disregard the *rest* entity by virtue of the fact that it contains source model entities that are not directly related to the MVC pattern. For example, both Java API's (i.e. `java` and `javax`) and utilities classes of JHotDraw have no direct influence on the architecture of the system.

5.2 ArgoUML



ArgoUML is an open source modeling tool that allows to create UML diagrams. It also provides advanced features, such as reverse engineering and code generation. This project is considered medium-sized and comprises between 99.5k and 159.5k lines of code. Despite an active community, in the official manual[4] of ArgoUML there is not an official description of the architecture of the system. As a starting point, we use the architecture shown in Figure 5.17 that is taken from the Software Tools and Techniques course given at the University of Auckland [Lut05]. Specifically, for this case study, for simplicity motivations, we model only `ArgoUML UI`, `GEF`, `Design Critics`, `UML Meta-Model` and `Code Generation`.

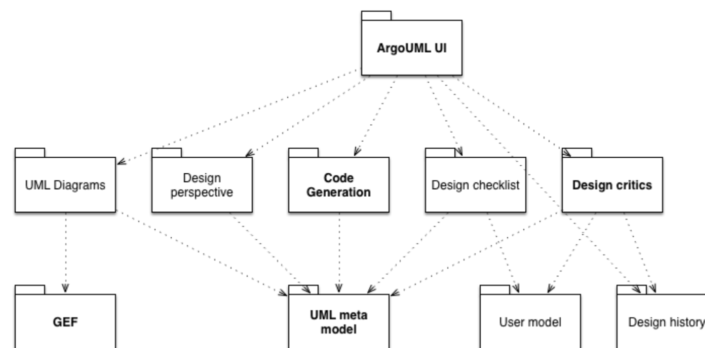


Figure 5.17. High-level architecture of ArgoUML.

The following sections are conceptually organized as in the previous case study.

5.2.1 High-level model entities

With the help of our tool, we investigate the design conformance of ArgoUML with the proposed architecture. After having uploaded the zipped project to our tool, the first step we perform is the creation of the five high-level model entities:

- ArgoUML UI
- Design Critics
- Code generation
- UML Meta Model
- GEF

After this step, we need to define the various mappings. In other words, we need to decide which source code entities are associated with which high-level model entity. We now discuss each one of them in detail.

[4] `http://argouml-stats.tigris.org/documentation/manual-0.32/`

ArgoUML UI

This entity collects the parts of the system that provide an infrastructure with menus, tabs and panes available for the other subsystems to fill with actions and contents. When a component wants to be placed into any of these UI elements it registers itself with the GUI subsystem using the appropriate method in `org.argouml.ui.GUI` class. When external subsystems supply their own user interface, they are split into two packages, e.g. `org.argouml.some-package` and `org.argouml.some-package.ui`. Figure 5.18 shows the dependencies between the main *GUI* entity and external subsystem.

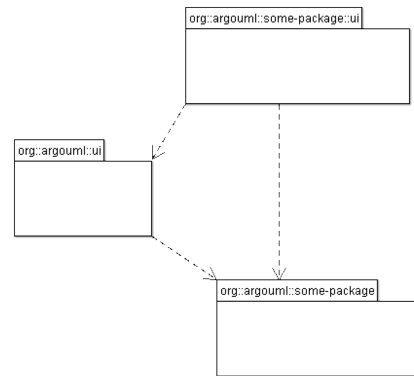


Figure 5.18. UI subsystem of ArgoUML.

Given the previous diagram we define a single mapping that matches all the namespaces ending with `ui` (i.e. `(.*)ui`). As it can be seen in Figure 5.19, the mappings defined before match a large portion of the system (i.e. more than twenty namespaces).

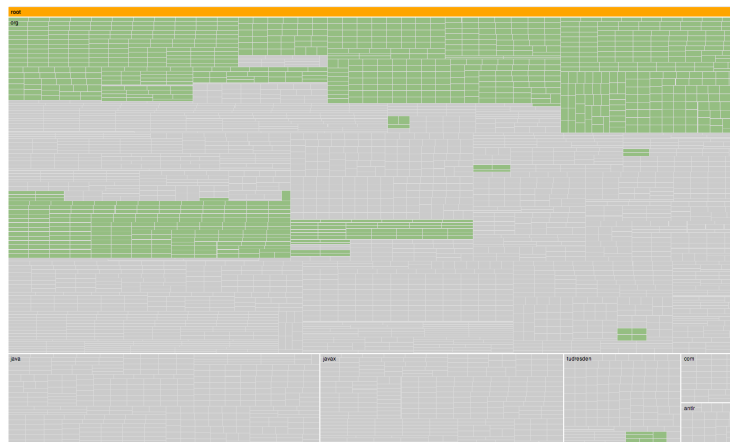


Figure 5.19. Coverage of the high-level entity UI.

Design Critics

Design Critics help to find artifacts in the model that do not obey simple design “rules” or “best practices”. While analyzing the system, we found critics in various places:

- `org.argouml.cognitive.critics`: contains basic critics, which are very general in nature (e.g. elements overlapping).
- `org.argouml.uml.cognitive.critics`: contains critics which are directly related to UML issues (e.g. classes with too many operations).
- `org.argouml.pattern.cognitive.critics`: contains critics related to patterns.

- `org.argouml.language.java.cognitive.critics`: contains critics strictly related to Java.

The well-structured namespace organization of ArgoUML, allow us to match all the critics with a single mapping. Specifically, we matched all the namespaces whose name ends in `critics` (i.e. `(.*)critics`).

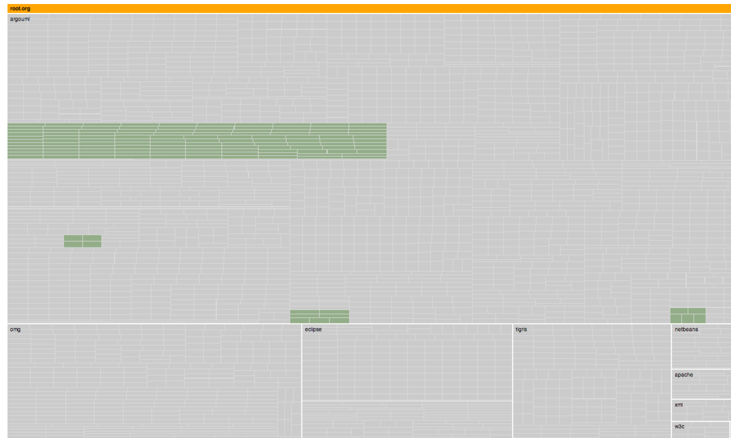


Figure 5.20. Coverage of the high-level entity Design Critics.

Code Generation

The code generation subsystem allows to automatically generate sources code files starting from a UML model. Language-specific implementations can found in sub-packages of `org.argouml.language`. Therefore we define a single mapping that matches all the namespaces ending with `generator` (i.e. `(.*)generator`). As a result (Figure 5.21) we match all the generators of the different languages available in ArgoUML: C#, PHP, Java, C++ and SQL.

Figure 5.21. Coverage of the high-level entity Code Generation.

UML Meta Model

The meta-model encapsulates the knowledge of what model repository is currently used and gives a consistent interface for manipulating data within the different repositories (e.g. MDR, EMF/UML2, NSUML). All the entities involved in this task are located in the `org.argouml.model` package and thus also in this case we create a single mapping to select all the entities. Specifically, we define a mapping that matches only the container namespace defined above (i.e. `org.argouml.model`).

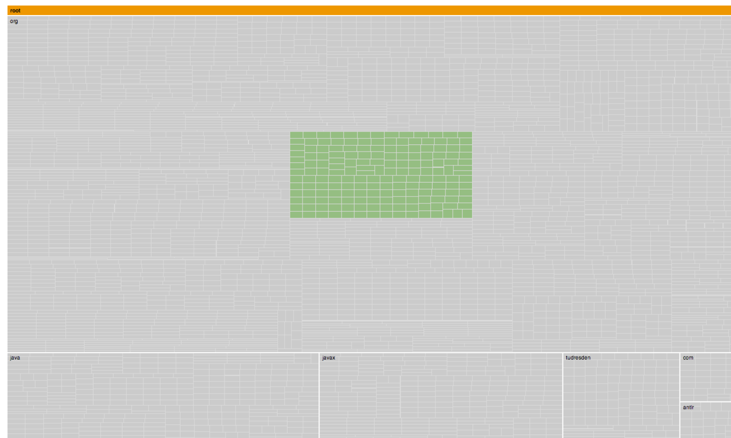


Figure 5.22. Coverage of the high-level entity UML Meta Model.

GEF

The Graph Editing Framework (GEF) provides reusable graph editing capabilities. More precisely, GEF is a graph editing library (similar to JHotDraw) that can be used to construct graph editing applications, i.e., applications that allows to draw structured and unstructured diagrams. This framework is enclosed inside `org.tigris.gef` package and thus, also in this last case, we create a single mapping to select all the entities involved. Specifically, we define a mapping that matches the namespace defined above (i.e. `org.tigris.gef`).

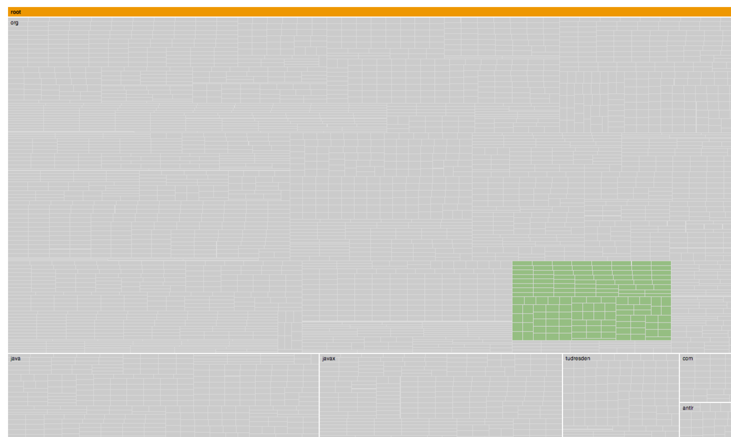
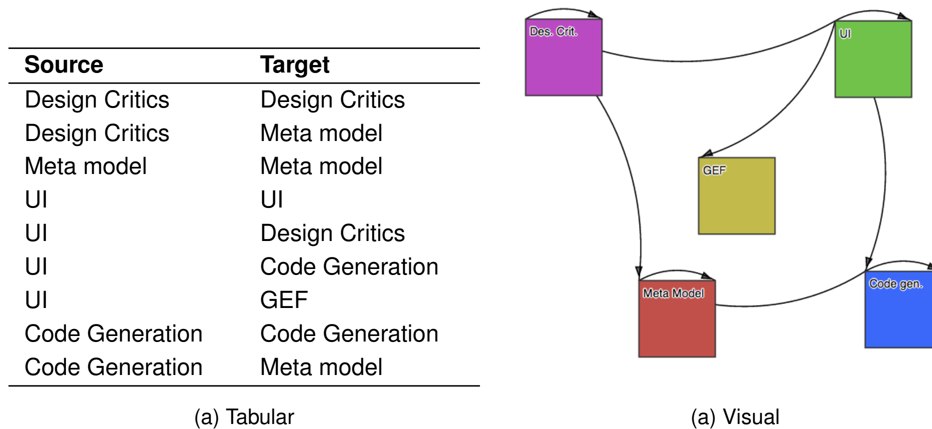


Figure 5.23. Coverage of the high-level entity GEF.

5.2.2 High-level model entities relations

At this point, we have all the high-level model entities in place. What we have to do now is to define how those entities relate to each other. To check the design conformance with respect to the proposed architecture, we define the various interactions as they are exposed in Figure 5.17. Specifically, we defined 8 relations as we show in Table 5.24a as well as in Figure 5.25a.



Source	Target
Design Critics	Design Critics
Design Critics	Meta model
Meta model	Meta model
UI	UI
UI	Design Critics
UI	Code Generation
UI	GEF
Code Generation	Code Generation
Code Generation	Meta model

(a) Tabular (a) Visual

Figure 5.25. Links between high-level model entities in ArgoUML.

5.2.3 Reflexion Model computation

The reflexion model shown in Figure 5.26 partially confirms the high-level model we created and thus also the initial architecture. Specifically, we can see that most of the relations are convergent. The divergent relations are mostly due to optimistic assumptions. For example, the relation that goes from the `UI` to the `Code Generation` was not present in the architecture of Figure 5.26 but can be found in the alternative architecture depicted in Figure 5.27 [GBTV13]. To sum up, we can safely assume that the high-level model we built is a good-enough representation of the real architecture.

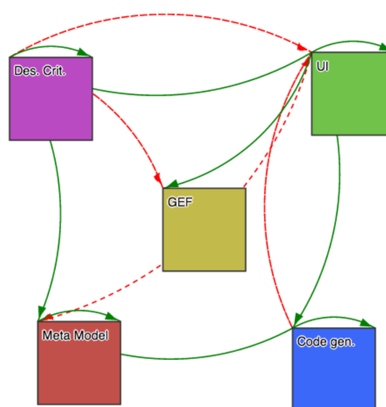


Figure 5.26. Reflexion Model of ArgoUML.

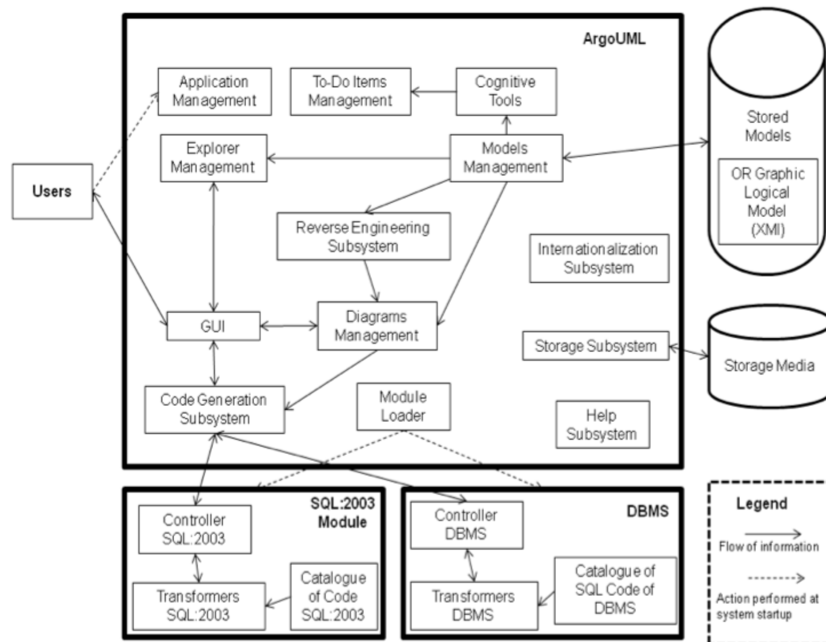


Figure 5.27. Alternative architecture of ArgoUML.

5.3 Reflections

The two case studies do not provide any result that can be considered statistically significant and we are aware of the fact that we still need to perform other experiments in order to really assess the quality of our approach. Despite the lack of a complete set of statistically relevant experiments, we believe that the results obtained are good indicator of the potential of our approach. Additionally, while we were using our tool we noticed possible features that can be added, that we will discuss as possible future work.

5.4 Summing Up

In this chapter we evaluated our approach by using our tool on two different software systems, JHot-Draw and ArgoUML. Despite not being statistically significant, these case studies served as starting point for future research and gave us indication about the potential of our method. Chapter 6 draws conclusions about the research conducted.

Chapter 6

Conclusions

This chapter concludes this thesis by reviewing the aspects that has been discussed so far. We first give a review of our approach and then we present the future work. Finally, we propose some extensions to our research.

6.1 Summary

Program comprehension is at the base of any process regarding software systems. To help developers in this complex task, Murphy et al. introduced the concept of Reflexion Model that was meant to bridge the gap between a mental-model a the developer has about a software system and the actual implementation [MNS01]. The main goal of our work is to create an evolution of the original Reflexion Model approach by putting the accent on the ease of use and immediateness. We now summarize our three contributions:

- **An intuitive approach for building, analyzing and refining reflexion models.** In Chapter 4, we presented an innovative and intuitive technique that allows developer to build and analyze Reflexion Model from a visual perspective. Being mostly visual, our approach enables virtually anyone to explore a software system and represents it by means of abstractions that can be then checked against the actual implementation.
- **The implementation of a web based application to help software engineers align their mental vision with an existing software system.** Starting from the technique we presented in Chapter 4, we developed a web based application that allows the user to take advantage of the Reflexion Model technique directly in their web browsers. The tool we developed provides an "out of the box" solution that can be employed not only by professional software engineers but, thanks to interactive and "playful" approach, it can be used also by undergraduate students.
- **A stand-alone plugin for the visualization of large software systems.** Standard treemaps become unreadable when dealing with large software systems. We decided to encapsulate the interactive squarified treemap we presented in Section 4.3.1 in a stand-alone extension of the `D3.js`[1] library. Thanks to our plugin, a developer can dive into the system progressively: starting from a global overview of the whole system to a detailed representation of the single entities.

[1] `http://d3js.org/`

In Chapter 5 we evaluated the Visual Reflexion Model approach using our tool. During the evaluation, we were able to understand interesting architectural aspects of both JHotDraw as well as ArgoUML. What we have learned about those two software systems using our tool gives us confidence in the usefulness of our idea.

6.2 Future Work

Our approach has been currently evaluated only by means of two case studies. In the future, we plan to perform other kinds of evaluations by including a small set of users in order to gather additional feedback and statistically relevant results. Additionally, during the test cases, we spotted a number of possible improvements and extensions to our approach:

- **Graphical evolution of the Reflexion Model**. To reduce the gap between the mental model and the actual implementation, a developer needs to progressively refine the high-level model and compute the resulting reflexion model. When the number of iteration starts to increase, it might become hard to recall every step. To overcome this limitation, we want to provide to the developer a storyboard containing a sequence of all the reflexion models produced so far. In this way, it becomes easier for the developer to understand if the reflexion model is evolving in the right direction or not.
- **Migrations of mappings between high-level model entities**. During our evaluation we discovered that sometimes mappings are created correctly but they are added to the wrong high-level model entity. In future releases we plan to introduce the ability to migrate mappings among high-level model entities, eliminating the need of rebuilding them from scratch.
- **Collaborative model building**. The myth of the "lone programmer" has already been disproved [Bro95]. For analogous motivations, also tasks related to program comprehension are nowadays carried out by multiple individuals rather than a single person. In the next future, we want to fully exploit the power of Firebase by allowing the developers to build high-level models in a collaborative fashion. Allowing multiple people to work on a single software system will ease the creative process as well as reducing the possible errors.
- **Reporting**. Our approach allows developers align their mental-model with the actual system. During this process, the developer learns more about the system under analysis by understanding its internal "mechanisms". The understanding process is however confined into the developer's mind and thus the knowledge acquired can diminish with time. To limit this effect, we want to introduce a reporting feature that will allow users to automatically generate an electronic report of their "comprehension process" which could be annotated with notes and comments.
- **Coverage Metrics**. Our visual approach allows the user to quickly get a sense of the high-level model being built. This way of considering the high-level model is however approximate and thus we plan to introduce explicit coverage metrics. Precisely, we want to give the developers accurate data about the portion of the source-model covered by the high-level model and information about its composition (e.g. number of entities, typologies, ...).
- **Explore others visualization techniques**. In Chapter 2 we presented different kinds of software visualization techniques and we highlighted the shortcomings of two-dimensional visualizations. Our tool currently visualize the source-model using a treemap but in the next future want to explore also three-dimensional visualizations. Specifically, we would like to exploit the power of CodeCity to allow the modelers to have a more complete (in terms of metrics) and engaging experience [WLR11].

- **Runtime Analysis**. The approach we have developed focuses on the static architecture of the system. Another useful view would be the runtime architecture where a developer can analyze and model how runtime-entities evolve and interact. Understanding the runtime architecture is important because it allows to analyze quality attributes such as security, performance and reliability. In this scenario, we would need to change the current representation of the source-model. Specifically, we can opt for a solution involving a graph representation where nodes correspond to objects, and edges correspond to relations between objects.

Appendices

Appendix A

Architecture

In this Chapter, we explain from a technical perspective all the key components of our tool.
From a high-level of abstraction, our system (Figure A.1) is composed by three main entities: the backend (Section A.1), the front-end (Section A.2) and a persistent storage (Section A.3).

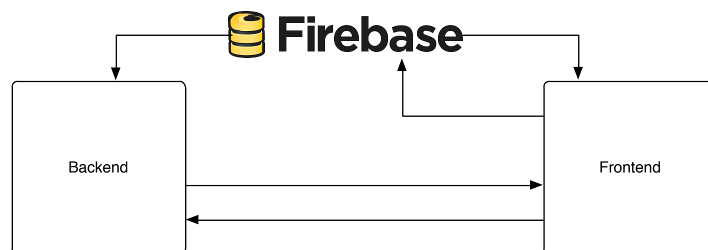


Figure A.1. High-level view of our tool.

In the following sections we first explain each entity in detail and then we move our focus on the interactions among them.

A.1 Backend

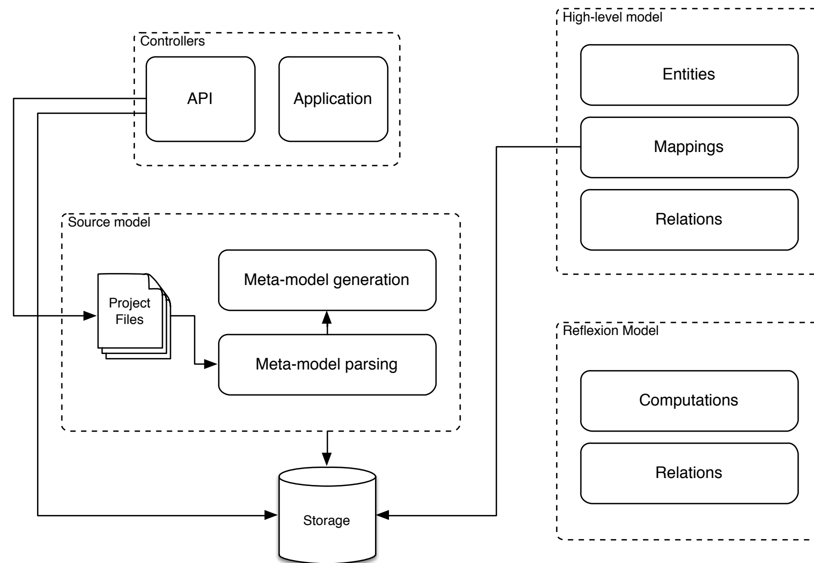


Figure A.2. High-level view of the backend.

Intuitively, the backend of our system is responsible for handling all the computations needed to compute the reflexion model and answering to the requests of the users. More precisely, the backend is organized in four sub-modules and a persistent storage (Figure A.2):

- **Controllers** are responsible for the interaction between our system and the user. In our implementation, we have two different controllers: one dedicated to serving web-pages and the other acting as an API. The API controller expects the user to send `GET` requests and as a response it will output a `JSON`. In Chapter B we provide a list of possible calls.

- **Source model** takes care of transforming the project uploaded by the user into a source model. To do so, the project is first decompressed, then it is given as an input to Verveinej which will generate an MSE file. The MSE file is then parsed with a script written in Scala which produces a flat list of all the elements contained in the FAMIX model. This list is then traversed using the Visitor pattern in order to collect all the relevant information and build the final source model.
- **High-level model** allows the user to define entities and the relations among them.
- **Reflexion model** is responsible of the representation and the computation of the reflexion model.
- **Storage** contains all the entities of the source model and allows the user to retrieve an source model entity given its ID and viceversa. This specific storage is semi-persistent and it will last until the termination of the server.

A.2 Front-end

As per definition, the front end is responsible for displaying results and collecting inputs from the user. In our case, the user has five different views (Table A.1).

URI	Description
/ - root	allow the user to select a step and allows to upload a file either using a dialog box or using a more fashionable drag-and-drop approach.
`/simplenavigator`	allows the creation of the high-level model entities and of the mappings
`/linker`	creates links between high-level model entities and displays the reflexion model
`/inspector/link/:from/:to`	allows the inspection of the relations between mappings
`/inspector/relation/:from/:fQuery/:to/:tQuery`	allows the inspection of the relations between source-model entities

Table A.1. List of views

A.3 Firebase (Storage)

All the information regarding the high-level entities, as well as their connections, are stored in the cloud.

Specifically, as we show in Figure A.3, our storage contains a list of (high-level) entities. Each entity has a global color (as we described in Section 4.3.1), a name and a list of queries. A query contains several attributes defining its visibility (`active`, `saved`, `visible` and `color`) and composition in terms of source-model entities contained (`included`).

The technology we picked is called Firebase[1] and among its features it includes a realtime data storage and synchronization API with support both for Java as well as HTML5.

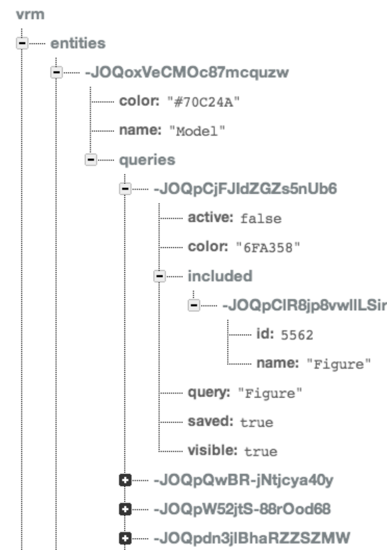


Figure A.3. Data representation in Firebase

[1] `https://www.firebase.com/`

A.4 Relations

In this section we briefly explain the relations among the components of our architecture.

- **From the Frontend to the Backend (and viceversa).** The frontend queries the backend mainly for retrieving webpages and to perform computations involving the source model.
- **From the Frontend to Firebase.** The frontend saves to Firebase every step performed by the user including the mappings created as well as the relations defined among the high-level entities.
- **From Firebase to the Backend.** Firebase pushes information regarding the mappings to the backend that will then take care of parsing the data received.
- **From Firebase to the Frontend.** When a modeling session is resumed, Firebase pushes information to the Frontend so that the visualizations can be rendered accordingly.

Appendix B

API

Our tool provides an application programming interface (API) that allows the users to query the back-end using standard HTTP methods (e.g., GET, PUT, POST, or DELETE). All the replies of the server will be encoded using the JSON[1] format. We now describe in detail the methods of the API and we give a concrete example for each one of them.

Description: Returns the list of entities for a given type of mapping and query
URI: `/api/models/:mapping/:search/:color`
Method: `GET`
Parameter - `mapping`: type of mapping to be used (name or namespace)
Parameter - `search`: query to be executed on the mapping
Parameter - `color`: color of the mapping in the hexadecimal format (e.g. `DEDEDE`)
Media type: `JSON`
Response structure:

```
{
   "query": (String),
   "included":[
      {
         "name": (String),
         "id": (Integer),
         "color": (String)
      },
     ...
   ]
}
```

We now give a concrete example:

[1] `http://json.org/`

URI: `/api/models/name/(.*)Event/70C24A`
Response structure:


```
{
   "query":"(.*)Event",
   "included":[
      {
         "name":"Event",
         "id":3739,
         "color":"70C24A"
      },
    ...
   ]
}
```

Description: Computes the reflexion model for the current session
URI: `/api/validate`
Method: GET
Media type: JSON
Response structure:


```
{
   "ok":[
      {
         "source": (String),
         "target": (String),
         "type": (String)
      },
     ...
   ],
   "extra":[
     {
        "source": (String),
        "target": (String),
        "type": (String)
     },
    ...
   ],
   "missing":[
       {
          "source": (String),
          "target": (String),
          "type": (String)
       },
      ...
   ]
}
```

We now give a concrete example:

URI: `/api/validate`
Response structure:

```
{
    "ok":[
        {
            "source":"UI",
            "target":"Des. Crit.",
            "type":"ok"
        },
        ...
    ],
    "extra":[],
    "missing":[
        {
            "source":"Code gen.",
            "target":"UI",
            "type":"missing"
        },
        ...
    ]
}
```

Description: Computes the relations between the mappings contained in two high-level model entities
URI: `/api/link/:from/:to`
Parameter - `from`: high-level model entity calling
Parameter - `to`: high-level model entity being called
Method: `GET`
Media type: `JSON`
Response structure:

```
{
    "links":[
        {
            "source": (String)
            "target": (String),
            "type": (String)
        },
      ...
    ]
}
```

We now give a concrete example:

URI: `/api/link/UI/Des.%20Crit.`
Response structure:

```
{
   "links":[
      {
         "source":"(.∗)ui",
         "target":"(.∗)critics",
         "type":"init"
      }
   ]
}
```

Description: Computes the relations between the source model entities contained in two mappings
URI: `/api/relation/:from/:fQuery/:to/:tQuery`
Parameter - `from`: high-level model entity calling
Parameter - `fQuery`: mapping of the high-level model entity calling
Parameter - `to`: high-level model entity being called
Parameter - `tQuery`: mapping of the high-level model entity being called
Method: `GET`
Media type: `JSON`
Response structure:

```
{
   "relations":[
      {
         "source": (String),
         "sourceid": (Integer),
         "sourceclass": (String),
         "target": (String),
         "targetid": (Integer),
         "targetclass": (String)
      },
     ...
   ]
}
```

We now give a concrete example:

URI: `/api/relation/UI/(.∗)ui/Des.%20Crit./(.∗)critics`
Response structure:

```
{
   "relations":[
      {
         "source":"initMenuCritique",
         "sourceid":156549,
```

```
            "sourceclass":"GenericArgoMenuBar",
            "target":"ActionOpenCritics",
            "targetid":114494,
            "targetclass":"ActionOpenCritics"
        },
      ...
    ]
}
```

Bibliography

[BE96] T. Ball and S.G. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, Apr 1996.

[Ber05] De Giacomo Berardi, Calvanese. Reasoning on {UML} class diagrams. *Artificial Intelligence*, 168(1-2):70–118, 2005.

[BHB99] Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster. Linux as a case study: Its extracted software architecture. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 555–563. ACM, 1999.

[BHVW00] Mark Bruls, Kees Huizing, and Jarke J Van Wijk. Squarified treemaps. In *Data Visualization 2000*, pages 33–42. Springer, 2000.

[Bro83] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International journal of man-machine studies*, 18(6):543–554, 1983.

[Bro95] Frederick P. Brooks, Jr. *The Mythical Man-month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[CVDK07] Mauro Cherubini, Gina Venolia, Rob DeLine, and Andrew J. Ko. Let's go to the whiteboard: How and why software developers use drawings. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, pages 557–566. ACM, 2007.

[DH07] Uri Dekel and James D. Herbsleb. Notation and representation in collaborative object-oriented design: An observational study. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 261–280. ACM, 2007.

[DN90] Lionel E Deimel and J Fernando Naveda. Reading computer programs: Instructor's guide to exercises. Technical report, DTIC Document, 1990.

[GBTV13] Aniruddha Guha Biswas, Raveesh Tandon, and Anurika Vaish. A case tool evaluation and selection methodology. *International Journal of Strategic Information Technology and Applications*, 4(2):48–60, April 2013.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[JS91] Brian Johnson and Ben Shneiderman. Tree-maps: A space-filling approach to the visualization of hierarchical information structures. In *Proceedings of the 2nd Conference on Visualization '91*, VIS '91, pages 284–291. IEEE Computer Society Press, 1991.

[Kai01] Wolfram Kaiser. Letters to the Editor. `http://www.javaworld.com/article/2077913/letters-to-the-editor.html`, 2001.

[KM99] Claire Knight and Malcolm Munro. Comprehension with [in] virtual environment visualisations. In *Proceedings of the IEEE 7th International Workshop on Program Comprehension*, CASCON '93, pages 4–11. IEEE, 1999.

[Kri97] R.L. Krikhaar. Reverse architecting approach for complex systems. In *Proceedings of the 5th International Conference on Software Maintenance*, ICSM '97, pages 4–11, Oct 1997.

[Let86] Stanley Letovsky. Cognitive processes in program comprehension. In *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*, pages 58–79, Norwood, NJ, USA, 1986. Ablex Publishing Corp.

[Lut05] Christof Lutteroth. Argo. University Lecture, Software Tools and Techniques, 2005.

[MKPW05] Kim Mens, Andy Kellens, Frédéric Pluquet, and Roel Wuyts. The intensional view environment. In *Proceedings of the 5th International Conference on Software Maintenance*, ICSM '97, pages 81–84, 2005.

[MN97] Gail C Murphy and David Notkin. Reengineering with reflexion models: A case study. *IEEE Computer*, 30(8):29–36, 1997.

[MNS01] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380, 2001.

[MTW93] Hausi A. Müller, Scott R. Tilley, and Kenny Wong. Understanding software systems using reverse engineering technology perspectives from the rigi project. In *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1*, CASCON '93, pages 217–226. IBM Press, 1993.

[O'B03] Michael P O'Brien. Software comprehension–a review & research direction. *Department of Computer Science & Information Systems University of Limerick, Ireland, Technical Report*, 2003.

[Pet09] Marian Petre. Insights from expert software design practice. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 233–242. ACM, 2009.

[Rei95] Steven P. Reiss. *The Field Programming Environment: A Friendly Integrated Environment for Learning and Development*. Kluwer Academic Publishers, Norwell, MA, USA, 1995.

[Riv00] Claudio Riva. Reverse architecting: An industrial experience report. In *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE'00)*, WCRE '00, pages 42–. IEEE Computer Society, 2000.

[SW93] J.T. Stasko and J.F. Wehrli. Three-dimensional computation visualization. In *In Proceedings of the 1993 IEEE Symposium on Visual Languages*, pages 100–107, Aug 1993.

[vMVH97] Anneliese von Mayrhauser, A. Marie Vans, and Adele E. Howe. Program understanding behaviour during enhancement of large-scale software. *Journal of Software Maintenance*, 9(5):299–327, September 1997.

[WHF93] Colin Ware, David Hui, and Glenn Franck. Visualizing object oriented software in three dimensions. In *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1*, CASCON '93, pages 612–620. IBM Press, 1993.

[WLR11] R. Wettel, M. Lanza, and R. Robbes. Software systems as cities: a controlled experiment. In *Proceedings of 33rd International Conference on Software Engineering*, ICSE 2011, pages 551–560, May 2011.

[WY96] Steven Woods and Qiang Yang. The program understanding problem: Analysis and a heuristic approach. In *Proceedings of the 18th International Conference on Software Engineering*, ICSE '96, pages 6–15. IEEE Computer Society, 1996.

[YM98] P. Young and M. Munro. Visualising software in virtual reality. In *Proceedings of the 6th IEEE International Workshop on Program Comprehension*, pages 19–26, Jun 1998.