Reverse Engineering Techniques for Lisp Systems

by
Adrian Dozsa

Diploma Thesis

Faculty of Automatics and Computer Science of the
"Politehnica" University of Timişoara

Timişoara,
September 2007

Advisors:
Dr. Ing. Tudor Gîrba
Conf. Dr. Ing. Radu Marinescu

Lisp is a programmable programming language.

John Foderaro

Abstract

Reverse engineering software systems, especially large legacy systems, is a difficult task, because of the sheer size and complexity of the systems. Many approaches have been developed to analyze software systems written in different languages. These approaches employ vary techniques like metrics or visualizations, and typically rely on parsing the system and on extracting a model that conforms to a meta-model.

However, no existent meta-model could fulfill the requirements for analyzing Lisp systems, so we developed the FAMIX-Lisp meta-model, as an extension of an existing meta-model. Our FAMIX-Lisp meta-model extends the initial meta-model with capabilities to model Lisp systems by adding new entities that are unique to Lisp, like Macros and CLOS entities.

Software visualization has been widely used by the reverse engineering research community during the past two decades, becoming one of the major approaches in reverse engineering.

In our thesis we also propose a set of new visualizations for Lisp systems, developed to underline the differences of the language and to help understand and browse complex Lisp systems, namely:

- The Class-Method Relation View is a visual way of supporting the understanding of the relation between classes and methods in a object-oriented Lisp program;
- The Class Types View is a visual way of identifying different types of classes, based on their structure (the attributes to methods ratio);
- The Programming Style Distribution View is a visual way of identifying the programming paradigm used in a program and their distribution over the system's packages;
- The Generic Concerns View is a visual way of identifying different cross-cutting concerns in a system by visualizing the spread of generic functions in the system.

The target of our views is to visualize very large Lisp systems, for which we want to obtain an initial understanding of their structure and their properties, which helps to guide software developers in the first steps of the reverse engineering process of an unknown system.

Acknowledgments

I am grateful to Radu Marinescu for all his support and for believing in me. Thanks for introducing me to research and for helping me find my direction, at least for this thesis. All those long talks were surely fascinating and fruitful. Your courses were the most interesting and exciting ones; I will miss them.

Special thanks to Tudor Gîrba (a.k.a. Doru) for making this thesis possible. It was nice to stay next to your office and to have you always available for no matter what problem I had. I will never forget all the interesting and useful discussions we had during my stay in Bern. Combining Chinese food and design, visualization or Mac talks was a lot of fun.

I thank Oscar Nierstrasz and the CHOOSE board for giving me the opportunity to work at the Software Composition Group and the SCG members for making me feel as part of the group (Oscar, Doru, Orla, Marcus, Adrian K., Adrian L., Lukas and David). A special thanks for Marcus and Adrian for having me in and for showing me around Bern.

I also like to thank Marius Minea for his encouragement and interest in my ideas, even if they weren't in his interest area. That is what a real professor does.

I thank John McCarthy for creating "the greatest single programming language ever designed" (Alan Kay), Lisp, which makes programming so much more fun.

Many thanks to all my friends from high-school (the renowned 12E class) and from faculty (Gabi, Paul, Elisa, Alex, Cezar, Cristina, Anda, Cosmina, Adriana, Eugen, Voda, Teleman, Dusco, Petri, Vali, Ioana, Diana, Stef, Calin, Anca, Dumi, Radu, Cosmin Dan and all the others my memory forgets) for just being my friends.

I would like to warmly thank my parents for all their love and for their unconditional support. I owe them so much for trusting me and encouraging me in all my choices. It was great to know that you were always standing by me no matter what I did. I also thank my brothers, Andrei and Cristi, for all the good times we had during our childhood and also student-hood. I don't know how I could go through all those long nights before the exams without a study partner.

My most special thanks are for Alina for being in my life, loving and encouraging me no matter what. You made me understand that every man has two sides: a rational one and an emotional one. Also thanks to her parents for having me into their home and for all the barbecues we had.

Timişoara, Adrian Dozsa
September 2007

Contents

1 Introduction **1**
- 1.1 Context . . . 1
- 1.2 Contributions . . . 2
- 1.3 Organization of the Thesis . . . 3

2 Reverse Engineering Techniques **5**
- 2.1 Definitions . . . 5
- 2.2 Modeling Techniques . . . 9
  - 2.2.1 The FAMIX Meta-model . . . 10
  - 2.2.2 Unified Modeling Language (UML) . . . 12
  - 2.2.3 Discussion . . . 13
- 2.3 Visualization Techniques . . . 13
  - 2.3.1 Polymetric Views . . . 15
  - 2.3.2 SHriMP Views . . . 17
- 2.4 Metrics . . . 19
- 2.5 Reverse Engineering Environments . . . 19
  - 2.5.1 Moose . . . 20
  - 2.5.2 iPlasma . . . 20
  - 2.5.3 Rigi . . . 21

3 Why Lisp? **23**
- 3.1 A Lisp Overview . . . 23
  - 3.1.1 Where it all started . . . 23
  - 3.1.2 General features . . . 24
  - 3.1.3 Myths and Legends . . . 27
- 3.2 Why is Lisp Different . . . 27
  - 3.2.1 Macros . . . 28
  - 3.2.2 Common Lisp Object System (CLOS) . . . 30
- 3.3 Summary . . . 32

4 The FAMIX-Lisp Meta-model **33**
- 4.1 Extending the FAMIX Meta-model . . . 33
  - 4.1.1 Adding Macros . . . 35
  - 4.1.2 Adding support for CLOS . . . 37

4.2 The FAMIX-Lisp Meta-model . . . 41
4.2.1 The Lisp Core . . . 43
4.2.2 The CLOS Core . . . 44
4.3 Examples . . . 44

5 Visualization Techniques for Lisp Systems 49
5.1 Introduction . . . 49
5.2 Polymetric views and Lisp . . . 50
5.3 The Class-Method Relation View . . . 52
5.4 The Class Types View . . . 55
5.5 The Programming Style Distribution View . . . 57
5.6 The Generic Concerns View . . . 60

6 Tool Support 63
6.1 ModeLisp: Lisp Model Extractor . . . 63
6.2 MoosLi: a Lisp plugin for Moose . . . 64
6.3 Visual browsers . . . 66

7 Case Studies 69
7.1 Overview . . . 69
7.2 Case study 1: SBCL . . . 70
7.3 Case study 2: Lisa . . . 77

8 Conclusions 83

List of figures 86

List of tables 87

Bibliography 93

Chapter 1

Introduction

"Take the first step in faith, you don't have to see the whole staircase, just take the first step."
Martin Luther King Jr.

1.1 Context

Reverse engineering software systems, especially large legacy systems, is technically difficult, because they typically suffer from several problems, such as developers no longer available, outdated development methods that have been used to write the software, outdated or completely missing documentation, and in general a progressive degradation of design and quality.

Reverse engineering existing software systems has become an important problem that needs to be tackled. It is the prerequisite for the maintenance, reengineering, and evolution of software systems. Since an unwary modification of one part of a system can have a negative impact, e.g. break, other parts of the system, one needs first to reverse engineer, e.g. have an informed mental model of the software, before the software system can be modified or reengineered.

The goal of a person who is reverse engineering a software system is to build progressively refined mental models of the system to be able to make informed decisions regarding the software. While this is not a complex problem for small software systems, where code reading and inspection is often enough, in the case of legacy software systems which tend to be large hundreds of thousands or millions of lines of poorly documented code are no exception this becomes a hard problem, because of their sheer size and complexity and because of the problems afflicting such systems [Par94]. In order to build a progressively refined mental model of a software system, the reverse engineer must gather information about the system which helps him in this process.

Reengineering large industrial software systems is impossible without appropriate tool support. First of all, there is the scalability issue (millions of lines of code) but there is also the extra complexity of supporting and combining multiple tools with a wide variety of tasks (standard forward engineering techniques must be combined with reverse- and re-engineering facilities).

To be able to reason about software systems, tools need a common information base, a repository, that provides them with the information required for reengineering tasks. The properties of the repository, and thus of the complete environment, are highly influenced by the meta-model

that describes what and in which way information is modelled.

In the reengineering research community several meta-models exist that model the software itself. They are aimed at procedural languages, object-oriented/procedural hybrid languages and systems with multiple paradigms. Most of the environments and associated meta-model focused on mainstream programming languages, like Java or C++, and some claimed that their work is language-independent. But all of the existent work fails to fully model complex Lisp systems because of the major differences this language presents.

Software visualization has been widely used by the reverse engineering research community during the past two decades. Koschke reports that 80% of interviewed researchers consider visualizations as being important or absolutely necessary in software reverse engineering [Kos03]. Recently many new visualizations have been developed, focusing on the changes of a system during time, on class hierarchies and class structure visualizations or on runtime information visualization. All these visualizations have the same goal: making it easier to understand complex coherences. The enormous interest in visualization as an aid for reverse engineering and problem detection can also be inferred from the large number of tools that have been developed for this purpose.

1.2 Contributions

The goal of this thesis is to develop reverse engineering techniques for analyzing Lisp systems. While the choice of Lisp seems strange, it is justified by the fact that Lisp is one of Computer Science's "classical" languages, based on ideas that have stood the test of time. On the other hand, it's a thoroughly modern, general-purpose language whose design reflects a deeply pragmatic approach to solving real problems as efficiently and robustly as possible. As one of the earliest programming languages, Lisp pioneered many ideas in computer science, including the if/then/else construct, tree data structures, recursive function calls, dynamic memory allocation, garbage collection, first-class functions, lexical closures, interactive programming, incremental compilation, meta-programming and dynamic typing. Lisp has a lot of unique and powerful features, some of them can't be found in any other languages, that will be studied in more detailed in this thesis.

Modeling Lisp systems is different from other languages. First of all the main reason is that Lisp is a multi-paradigm language, it combines more programming paradigms and styles in one language. But even in the context of object-oriented programming languages, for example, Lisp sets itself apart from other languages with different language entities and new ways of combining those entities, as well see it later. From the meta-modeling point of view the main difficulties are Macros and CLOS. Macros are a unique feature that can't be found in any other languages (they are very different from C macros) and CLOS is a very different object system from other object-oriented languages, like Java, C++ or even Smalltalk.

In this thesis we propose the FAMIX-Lisp meta-model, as an extension of the FAMIX meta-model [DTD01]. Our FAMIX-Lisp meta-model extends the FAMIX meta-model with capabilities to model Lisp systems by adding some new entities, like Macros and CLOS entities. FAMIX-Lisp is an extension and not a modification of the FAMIX meta-model, meaning that the our meta-model is still compatible with the FAMIX meta-model.

To validate the ability of the FAMIX-Lisp meta-model to model large Lisp systems we developed two tools that implement the FAMIX-Lisp meta-model. First we have a model extractor

(*ModeLisp*) that extract FAMIX-Lisp models from Lisp systems and then we have a Lisp plugin (*MoosLi*) for the MOOSE environment [NDG05], that can import FAMIX-Lisp models and browse the model and apply different analysis and visualizations on them.

We applied existing visualization approaches [Lan03] on some Lisp system and present here the results of the experiment. We also propose a set of novel visualization for Lisp systems, developed to underline the differences of the language and to help understand and browse complex Lisp systems:

- the Class-Method Relation View
- the Class Types View
- the Programming Style Distribution View
- the Generic Concerns View

The target of our views is to visualize very large Lisp systems, for which we want to obtain an initial understanding of their structure and their properties. This information is useful for identifying the parts of a subject system which need to be further analyzed, and to obtain an overall view that reduces the complexity inherent in such large systems. Our views can answer questions about the overall structure of a system, about the detection of particular software artifacts, and provide a first impression of a subject system, which helps to guide software developers in the first steps of a reverse engineering process of an unknown system.

To validate our approach we applied our newly developed techniques on several medium to large Lisp case studies and we'll present the result of our experiments in detail in Chapter 7.

1.3 Organization of the Thesis

This dissertation is structured as follows:

Chapter 2 introduces the problem domains of reverse engineering, software modelling and software visualization.

Chapter 3 introduces the Lisp languages and why Lisp is different from other programming languages.

Chapter 4 presents the process adding support for Lisp to the FAMIX Meta-model and the developed FAMIX-Lisp Meta-model.

Chapter 5 introduces four new visualization specially tailored for Lisp.

Chapter 6 presents the tools we developed to support our work.

Chapter 7 provides two case studies for our work.

Chapter 8 concludes by summarizing the main contributions of our work and give also an outlook on possible future work in this research field.

Chapter 2

Reverse Engineering Techniques

"Programs must be written for people to read and only incidentally for machines to execute."
Hal Abelson and Gerald Jay Sussman

2.1 Definitions

Chikofsky and Cross [CCI90] define reengineering as follows:

> "*Reengineering* is the examination and the alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form."[CCI90]

As stated by the definition, reengineering consists of two main activities, namely the examination and the alteration of a subject system. More formal terms for these activities are reverse engineering and forward engineering:

> "*Reverse engineering* is the process of analyzing a subject system to (i) identify the systems components and their relationships and (ii) create representations of the system in another form or at a higher level of abstraction."[CCI90]

> "*Forward engineering* is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system."[CCI90]

The main difference between forward engineering and reengineering is that reengineering starts from an existing implementation. Consequently, for every change to a system the reengineer must evaluate whether the system need to be restructured (or refactored) or if they should be implemented anew from scratch. According to Chikofsky and Cross, restructuring generally refers to source code translation, but it may also entail transformations at the design level. This is their definition:

> "*Restructuring* is the transformation from one representation form to another at the same relative abstraction level, while preserving the systems external behavior."[CCI90]

Refactoring is merely a special kind of restructuring, namely within an object-oriented context and focused on the level of code. Typical goals of refactoring are to improve the simplicity, understandability, flexibility or performance. In his catalog of refactorings Martin Fowler defines it as follows:

> "*Refactoring* is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure." [FBB+99]

The reengineering life-cycle

The process to get from a legacy system to a reengineered system is described by Casais [Cas98] in a five step reengineering life-cycle that can be mapped to Figure 2.1:

1. Model capture (documenting and understanding the design of the legacy system),
2. Problem detection (identifying violations of flexibility and quality criteria),
3. Problem analysis (selecting a software structure that solves a design defect),
4. Reorganization (selecting and applying the optimal transformation of the legacy system),
5. Change propagation (ensuring the transition between different software versions).

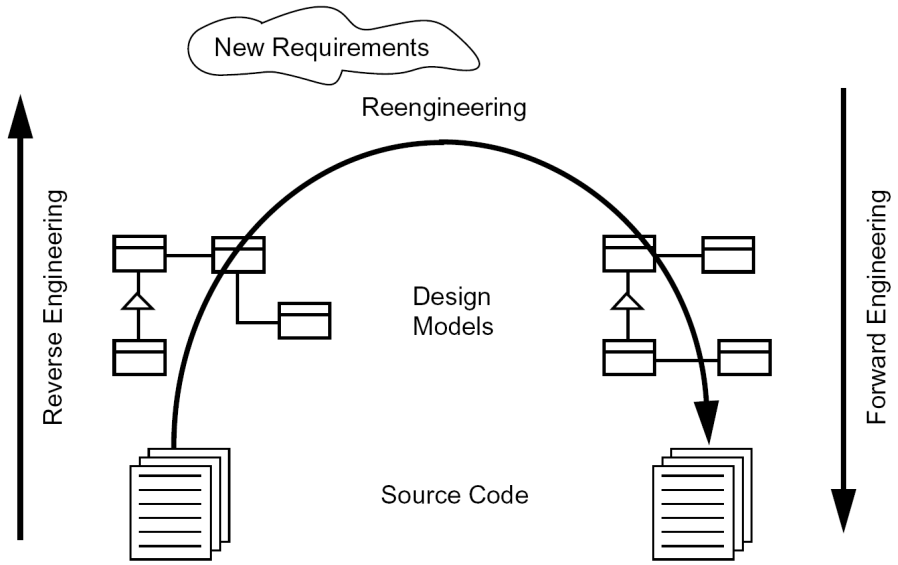

Figure 2.1: The reengineering life-cycle

If forward engineering is about moving from high-level views of requirements and models towards concrete realizations, then reverse engineering is about going backwards from some concrete realization to more abstract models, and reengineering is about transforming concrete

implementations to other concrete implementations. In a typical legacy system, you will find that not only the source code, but the documentation and specifications are out of sync. Reverse engineering is therefore a prerequisite to reengineering since you cannot transform what you do not understand. You carry out reverse engineering whenever you are trying to understand how something really works. Normally you only need to reverse engineer a piece of software if you want to fix, extend or replace it. As a consequence, reverse engineering efforts typically focus on documenting software and identifying potential problems, in preparation for reengineering. In Figure 2.1 we see this illustrated.

Goals of Reverse Engineering

The goal of a person that is reverse engineering a software system is to build progressively refined mental models of the system [SFM99] to be able to make informed decisions regarding the software. While this is not a complex problem for small software systems, where code reading and inspection is often enough, in the case of legacy software systems which tend to be large hundreds of thousands or millions of lines of poorly documented code are no exception this becomes a hard problem because of their sheer size and complexity, and because of the problems afflicting such systems [Par94]. In order to build a progressively refined mental model of a software system, the reverse engineer must gather information about the system which helps him in this process.

Chikofsky and Cross state that "the primary purpose of reverse engineering a software system is to increase the overall comprehensibility of the system for both maintenance and new development" [CCI90]. They list six key reverse engineering objectives:

1. Cope with complexity
2. Generate alternate views
3. Recover lost information
4. Detect side effects
5. Synthesize higher abstractions
6. Facilitate reuse

Before starting a reverse engineering process it is therefore essential to decide which primary goals to pursue and which ones are only of secondary importance.

Approaches to Reverse Engineering

There are many approaches to reverse engineering software systems, such as:

- reading the existing documentation and source code. Various people have investigated code inspection, code reading, and code review practices [DDN02]. Using this approach is difficult when the documentation is obsolete, incorrect or not present at all. Reading the source code is a widely used practice, but does not scale up, as reading millions of lines of code would take weeks or months without necessarily increasing the understanding of the system by the reader. Moreover, at the beginning of a reverse engineering process one does not seek detailed information, but rather wants to have a general view of the system.

- running the software and/or generate and analyze execution traces. The use of dynamic information, e.g., information gathered during the execution of a piece of software, has also been used in the context of reverse engineering [RD99, Gre07], but has drawbacks in terms of scalability (traces of a few seconds can become very big) and interpretation (thousands of message invocation can hide the important information one is looking for).

- interviewing the users and developers. This can give important insights into a software system, but is problematic because of the subjective viewpoints of the interviewed people and because it is hard to formalize and reuse these insights. Moreover, it can be hard to find developers that have been part of the development team over long periods of time and thus possess knowledge about a software systems complete lifetime.

- using various tools (visualizers, slicers, query engines, etc.) and techniques (visualization, clustering, concept analysis, etc.) to generate high-level views of the source code. Tool support is provided by the research community in various ways, and visualization tools like Rigi [Mül86] and ShrimpViews [SM95] are often used.

- analyzing the version history, as for example done by Gîrba [GÎ05]. Still a young research field, understanding the evolution of a piece of software is done using techniques like graph rewriting, visualization, concept analysis, clustering, and data mining. The insights gained are useful to understand the past of a piece of software and to possibly predict its evolution in the future.

- assessing a software system and its quality by using software metrics. Software metrics tools are used to assess the quality and quantity of source code by computing various metrics which can be used to detect outliers and other parts of interest, for example cohesive classes, coupled subsystems, etc.

Object-Oriented Reverse Engineering

Although the term "legacy system" is often associated with systems in assembler or procedural languages such as Fortran and Cobol, object-oriented systems suffer from similar problems. The Laws of Lehman [LB85, Leh96] tend to be true for systems in any language. This is supported by facts: object-oriented legacy applications exist even in relatively young languages such as Java [Duc97]. Furthermore, reengineering techniques are starting to become part of modern software development processes. Hence, also in that context reengineering techniques are relevant for systems implemented in languages other than the traditional COBOL, Fortran or C.

Apart from common legacy problems such as duplicated functionality and insufficient and outdated documentation, reengineering object-oriented languages presents its own set of problems [WH92]. We list here some of the most preeminent:

- Polymorphism and late binding make traditional tool analyzers like program slicers inadequate. Data-flow analyzers are more complex to build especially in presence of dynamically typed languages.

- Incremental class definition, together with the dynamic semantics of *self* or *this*, make applications more difficult to understand.

- Dynamically typed languages such as Smalltalk, on the one hand, make the analysis of applications harder because types of variables are implicit and tool support is needed to infer them. On the other hand, statically typed languages such as C++ and Java force the programmer to explicitly cast objects, which leads to applications that are less maintainable and require more effort to be changed.

Apart from the above list, common code-level problems occurring in object-oriented legacy systems are often due to misuse or overuse of object-oriented features, such as the misuse of inheritance or the violation of encapsulation.

2.2 Modeling Techniques

In order to specify the concept of *meta-modeling*, we consider the following definitions [BG01]:

> "*A model* is an abstraction of a system, that should be more simple. Most of the time, this means hiding technical details. A model represents the system it describes, and could be used in replacement to answer questions about it."

> "*A meta-model* defines the specification of an abstraction, i.e. one or several models. This specification defines a set of important concepts to define models, as well as the relationship between these concepts. A meta-model defines the vocabulary to be used for defining models."

Modeling is intended to design systems using a predefined set of concepts. Meta-modeling is intended to specify concept to be used for defining models. Meta-modeling introduces the flexibility required to define adapted means to software process requirements, in order to design and build systems. Two key expressions summarizing this approach are:

- "an attempt to describe the world", and
- "for a particular goal".

The second expression illustrates the fact that there will never be a "universal"meta-model to define all the software systems from embedded to worldwide ones. It is important to understand that a (meta)model has to be defined for a specific goal. Thus, there are a multitude of (meta)models. The need for dedicated means is required for providing appropriate solutions to requirements of a field of activitylike e-business, telecom, or embedded systems.

Meta-models and Software Reverse Engineering

To be able to reason about software systems, software engineers need a common information base, a repository, that provides them with the information required for reengineering tasks. The properties of the repository are highly influenced by the meta-model that describes what and in which way information is modelled. The meta-model not only determines if the right information is available to perform the intended reengineering tasks, but also influences issues such as scalability, extensibility and information exchange.

There are a number of existing meta-models for representing software. Several of those are aimed at object-oriented analysis and design (OOAD), the most notable example being the

Unified Modeling Language (UML) [Gro04]. However, these meta-models represent software at the design level. Reengineering requires information about software at the source code level. The starting point is the software itself demanding for a precise mapping of the software to a model rather than a design model that might have been implemented in lots of different ways. In the reengineering research community several meta-models exist that model the software itself. They are aimed at procedural languages (Bauhaus [CEK+00]), object-oriented/procedural hybrid languages (Datrix [HHL+00]) and systems with multiple paradigms (FAMIX [Tic01]). Most meta-models support multiple languages, either implicitly or explicitly.

2.2.1 The FAMIX Meta-model

This section introduces FAMIX [DTD01], a meta-model for modeling multiple hybrid procedural, object-oriented languages. The main goal is to support reengineering activities in a language-independent way. The aim was not to cover all aspects of all languages, but rather to capture the common features needed for reengineering activities, so tools can be easily reused for multiple target languages [Tic01].

The FAMIX meta-model models multiple languages. It defines a language-independent core, which allows tools to be reusable without adaptation over the supported languages. How languages are mapped to the core and which language specifics can be stored, is specified in language extensions (see Figure 2.2). There are multiple plug-ins for specific programming languages, like C++, Java, Smalltalk and Ada.

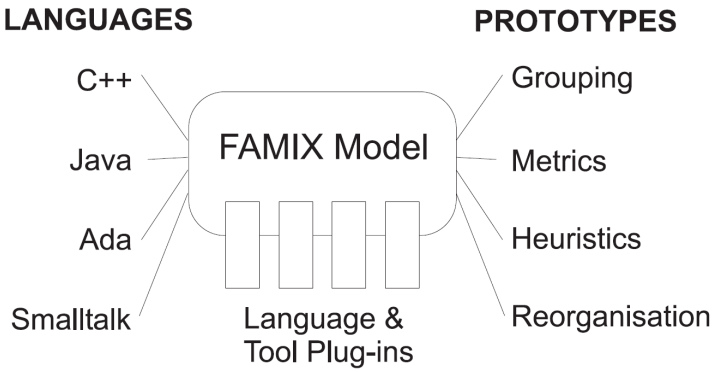

Figure 2.2: Conception of the FAMIX meta-model

The meta-model represents source code, at the program entity level as opposed to the abstract syntax tree level. The information allows one to perform structural analysis and dependency analysis. It supports metrics computation and heuristics. It does not support control flow analysis and the regeneration of source code from the model. The model stores, however, the location of the source code, allowing one to obtain additional information from the source code itself. A second reason to choose the program entity level was that more detailed information increases the size of models considerably which hampers scalability. Thirdly, the program entity level enables to abstract from language-specific details and thus obtaining a clean language-independent meta-model.

Figure 2.3 shows the core entities and relations. All basic elements of an object-oriented languages are present (Class, Method, Attribute). Furthermore, FAMIX models dependency information, such as method invocations (which method invokes which method) and attribute accesses (which method accesses which attribute).

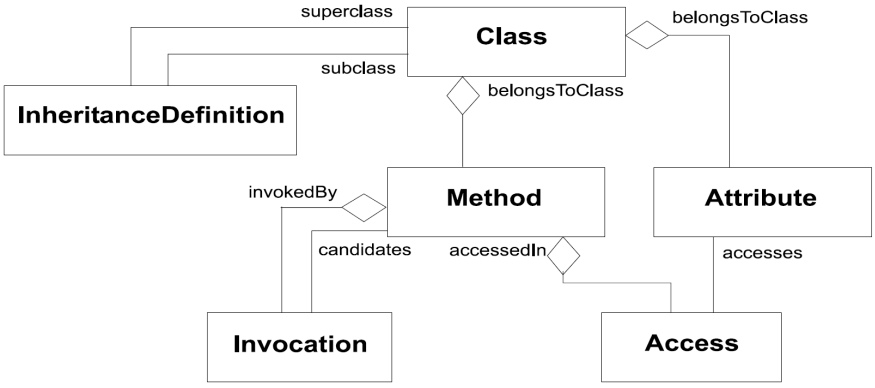

Figure 2.3: The core of the FAMIX meta-model

The complete meta-model is not restricted to the above elements. Additionally it also models different kinds of variables, functions and arguments.

There are multiple ways in which the FAMIX meta-model can be extended:

- *New model elements:* an extension can define new model elements.
- *New attributes to existing model elements:* existing elements can be extend to allow one to store additional information.
- *Annotations:* any model element can be annotated by attaching a property to it.

The following decisions in the design of FAMIX are relevant for the support of multiple languages:

- *Statically typed and dynamically typed languages:* Static type information is important to store, because it reveals important dependencies. If the information is not known, which is normally the case with dynamically typed languages such as Smalltalk, the information can be inferred or is left empty.
- *Multiple inheritance:* FAMIX supports multiple inheritance. This allows us to deal with single inheritance languages such as Smalltalk or Java, but also with multiple inheritance languages such as C++.
- *Pointer, array and other non-Class types:* FAMIX does not explicitly model pointer, primitive array and other primitive types, but provides a way to include the information into the model with the help of some special attributes.

An important part of the usability of the meta-model depends on how the actual programming languages are mapped to language-independent constructs. The goal was to treat as many concepts of different languages as possible uniformly. On the other hand, the model stores information about the mappings, because the semantic difference of a similar concept in different languages might be of interest for certain tools.

FAMIX uses the MSE, CDIF and XMI exchange formats for information exchange. The MSE format was chosen, because of its human readability and support for incremental loading. The entities are not nested into their scoping entity and simple relationships are stored as attributes of the entity it is linked to. Relationships are represented either as attributes of model elements for containment relationships, or as explicit entities for the other, mostly attributed, relationships. There is no specific relationship kind of elements.

In FAMIX, model elements can be referenced in two ways. Firstly, every model element has a unique identifier. Secondly, all elements that have an intrinsic unique name, i.e, all instances of Entity and its subclasses, can be uniquely identified by that name.

2.2.2 Unified Modeling Language (UML)

In the field of software engineering, the Unified Modeling Language (UML) [SMHP+04] is a standardized specification language for object modeling. UML is a general-purpose modeling language that includes a graphical notation used to create an abstract model of a system, referred to as a UML model.

UML is officially defined by the Object Management Group (OMG) by the UML meta-model, a Meta-Object Facility meta-model (MOF). Like other MOF-based specifications, the UML meta-model and UML models may be serialized in XMI. UML was designed to specify, visualize, construct, and document software-intensive systems.

UML helps you specify, visualize, and document models of software systems, including their structure and design. You can use UML for business modeling and modeling of other non-software systems too. UML is extensible, offering the following mechanisms for customization: profiles and stereotype.

UML has been a catalyst for the evolution of model-driven technologies, which include Model Driven Development (MDD), Model Driven Engineering (MDE), and Model Driven Architecture (MDA). By establishing an industry consensus on a graphic notation to represent common concepts like classes, components, generalization, aggregation, and behaviors, UML has allowed software developers to concentrate more on design and architecture.

UML defines thirteen types of diagrams, divided into three categories:

- *Structure Diagrams* emphasize what things must be in the system being modeled (include the Class Diagram, Object Diagram, Component Diagram, Composite Structure Diagram, Package Diagram, and Deployment Diagram);
- *Behavior Diagrams* emphasize what must happen in the system being modeled (include the Use Case Diagram, Activity Diagram, and State Machine Diagram);
- *Interaction Diagrams*, a subset of behavior diagrams, emphasize the flow of control and data among the things in the system being modeled (derived from the more general Behavior Diagram, include the Sequence Diagram, Communication Diagram, Timing Diagram, and Interaction Overview Diagram).

2.2.3 Discussion

Legacy systems exist in many languages. On top of that, many reengineering tasks are similar for multiple languages, especially within a single paradigm. Consequently, there is a vast potential for reuse over multiple similar languages. To be able to deal with multiple languages effectively it needs to be clearly defined how different languages are represented in a common way. Only in this way tools will be able to base common analysis on the meta-model and be sure that it provides the expected results for all supported languages. However, not many meta-models have elaborate multi-language support.

If only one language, hybrid or not, needs to be supported, the meta-model typically contains constructs of that language in a straightforward one-to-one mapping. It gets more complex if multiple languages are supported. The constructs of both languages are modelled either separately or using a common abstraction. Separate modeling is typical in the case of languages that have dissimilar paradigms. In such a case the meta-model is often constructed with separate, but connected sub-meta-models for every paradigm [LS99]. If the supported languages have the same or an overlapping paradigm, common constructs are often modelled with a single abstraction. The typical structure of such a meta-model is a language-independent core with multiple language extensions. For instance, a core could contain a Class abstraction which would allow class concepts in Java, Smalltalk and C++ uniformly, but the C++ class template would be modelled in the C++ language extension. Treating similar constructs in a similar way results in language independence and reuse of analysis code. On the other hand, treating them explicitly decreases problems with semantic differences.

Most of the environments focused on mainstream programming languages, like Java or C++, and some claimed that their work is language-independent. But all of the existent work fails to fully model complex Lisp systems. Lisp goes beyond other programming languages. We will see why in Chapter 3. And then in Chapter 4 will provide a solution, based on an existing meta-model.

2.3 Visualization Techniques

Software visualization is defined as "the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software."[SDBP98]. It is a specialization of information visualization, whose goal is to visualize any kind of abstract data, while in software visualization the sole focus lies on visualizing software.

Information Visualization. Information visualization is defined as "the use of computer-supported, interactive, visual representations of abstract data to amplify cognition." [CMS99]. It derives from several communities. Starting with Playfair, in 1786, the classical methods of plotting data were developed. In 1967, Jacques Bertin, a French cartographer, published his theory in the semiology of graphics. This theory identifies the basic elements of diagrams and describes a framework for their design. Edward Tufte published a theory of data graphics that emphasized maximizing the density of useful information [Tuf90, Tuf97]. Both Bertins and Tuftes theories have been influential in the various communities that led to the development of information visualization.

The goal of information visualization is to visualize any kind of data. Note that the above definition of information visualization does not necessarily imply the use of vision for perception: visualizing does not necessarily involve visual approaches, but any kind of perceptive approach (visual, auditive or tactile). It must be emphasized that most information visualization systems involve using computer graphics which render the data using 2D and/or 3D-views of the data. Applications in information visualization are so frequent and common, that most people do not notice them: examples include meteorology (weather maps), geography (street maps), geology, medicine (computer-aided displays to show the inner of the human body), transportation (train tables and metro maps), etc.

In short, information visualization is about visualizing almost any kind of data in almost any kind of way, while software visualization is about visualizing software.

The field of software visualization can be divided in two separate areas [SDBP98]:

1. *Program visualization* is the visualization of actual program code or data structures in either static or dynamic form. The visualizations presented in Chapter 5 belong to a sub-area of program visualization, namely static code visualization, because we visualize source code by using only information which can be statically extracted from the source code without the need to actually run the system.

2. *Algorithm visualization* is the visualization of higher-level abstractions which describe software. A good example is algorithm animation, which is the dynamic visualization of an algorithm and its execution. This was mainly done to explain the inner working of algorithms. In the meantime this discipline has lost importance, mainly because the advancement in computer hardware and the possibility to use standard libraries.

Software visualization and reverse engineering. Software visualization has been widely used by the reverse engineering research community during the past two decades [SDBP98]. Many of the approaches provide ways to uncover and navigate information about software systems. The graphical representations of software used in the field of software visualization, a sub-area of information visualization, have long been accepted as comprehension aids to support reverse engineering. Software visualization has become one of the major approaches in reverse engineering, Koschke reporting that 80% of interviewed researchers consider visualizations as being important or absolutely necessary in software reverse engineering [Kos03].

Recently many new visualizations have been developed. Some focus on the changes of a system during time [GÛ5] others on class hierarchies and class structure visualizations [Lan03] and again others on runtime information visualization [Gre07]. Several systems make use of the third dimension by rendering software in 3D [WL07]. All these visualizations have the same goal: making it easier to understand complex coherences.

The enormous interest in visualization as an aid for reverse engineering and problem detection can also be inferred from the large number of tools that have been developed for this purpose: Rigi [Mül86], SHriMP [MADSM01], CodeCrawler [LDGP05], Mondrian [MGL06], etc.

2.3.1 Polymetric Views

Polymetric views [LD03] are a lightweight software visualization technique enriched with software metrics information. Polymetric views help to understand the structure and detect problems of a software system in the initial phases of a reverse engineering process.

The polymetric view visualization uses two-dimensional displays to visualize object-oriented software. The nodes represent software entities or abstractions of them, while the edges represent relationships between those entities. For enriching this basic visualization method, there are rendered up to five metric measurements on a single node simultaneously: node size (width and height), node color and node position (X and Y coordinates). An actual visualization depends on three ingredients:

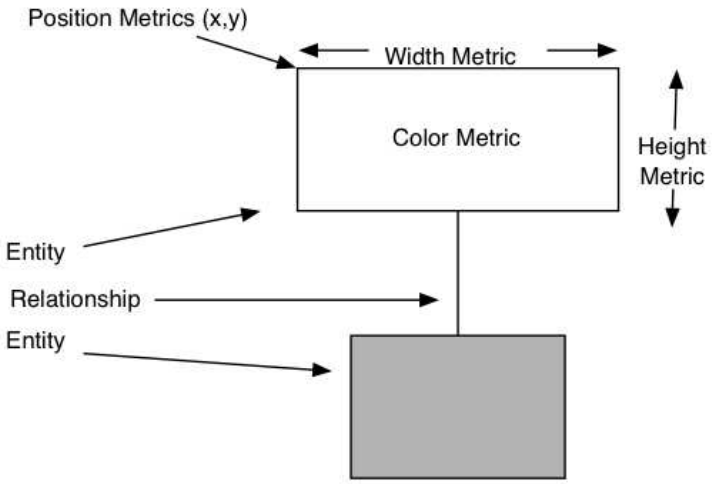

Figure 2.4: Basic elements of a polymetric view

- *A layout.* A layout takes into account the choice of the displayed entities and their relationships. Example of layouts are: tree, scatterplot, checker.

- *The metrics.* A visualization can include up to five metrics. The choice of the metrics heavily influences the resulting visualization, as well as its interpretation.

- *The entities.* Certain views are better suited for small parts of the system, while others can handle a complete large system. The reverse engineer must choose which parts or entities of the subject system he wants to visualize.

Depending on the applied polymetric view, the viewer can visually (e.g., by looking and interacting with the visualization) extract different kinds of information about the visualized system, i.e., information about the structure of hierarchies, about the size of classes and methods, about the use of attributes, etc.

The system complexity view. This view is based on the inheritance hierarchies of a subject system and gives clues on its complexity and structure. For very large systems, it is advisable to apply this view first on subsystems, as it takes quite a lot of screen space. The goal of this view is to classify inheritance hierarchies in terms of the functionality they represent in a subject system.

The system complexity view visualizes classes as nodes, while the edges represent inheritance relationships. The metrics used to enrich this view are NOA (the number of attributes of a class) for the width and NOM (the number of methods of a class) for the height. The color shade represents WLOC (the number of lines of code of a class).

In Figure 2.5 we have an example of the system complexity view of a Java project.

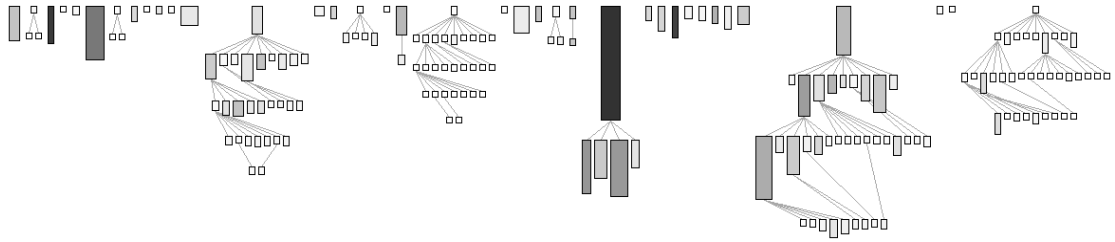

Figure 2.5: A complexity view of a Java project

This view is based on the inheritance hierarchies of a subject system and gives clues on its complexity and structure. For very large Systems, it is advisable to apply this view first on subsystems, as it takes quite a lot of screen space. The goal of this view is to classify inheritance hierarchies in terms of the functionality they represent in a subject system.

With the help of this view we can identity some bad design symptoms, tall and narrow classes with few attributes and many methods, deep or large hierarchies, standalone large nodes represent classes with many attributes and methods without subclasses, light flat nodes with a width to height ration of 1:2. For more details see [LD03].

Class blueprints. This paragraph introduces the concept of the class blueprint [DL05], a visual way of supporting the understanding of classes. A class blueprint is a semantically augmented visualization of the internal structure of a class, which displays an enriched call-graph with a semantics-based layout.

A class blueprint is structured according to layers that group the methods and attributes. The nodes representing the methods and attributes contained in a class are colored according to semantic information, i.e., whether the methods are abstract, overriding other methods, returning constant values, etc. The nodes vary in size depending on source code metrics information.

The layers support a call-graph notion in the sense that a method node on the left connected with another node on the right is either invoking or accessing the node on the right that represents a method or an attribute. In Figure 2.6, from left to right we identify the following layers: *initialization layer, external interface layer, internal implementation layer, accessor layer, and attribute layer.* The first three layers and the methods contained therein are placed from left to right according to the method invocation sequence, i.e., if method m1 invokes method m2, m2 is placed to the right of m1 and connected with an edge.

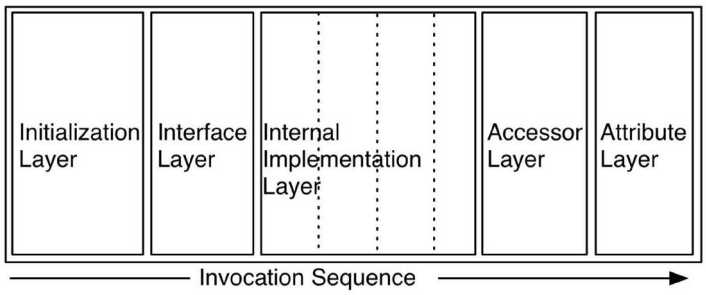

Figure 2.6: The decomposition of a class blueprint into layers

In Figure 2.7 we have an example of the class blueprint visualization of a Java class.

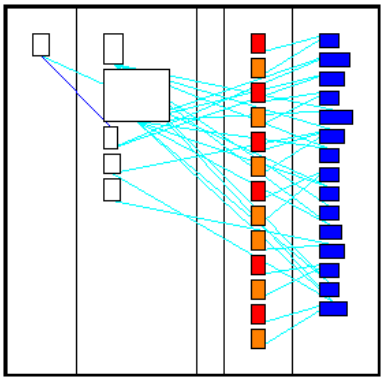

Figure 2.7: A blueprint visualization of a Java class

2.3.2 SHriMP Views

The SHriMP (Simple Hierarchical Multi-Perspective) tool provides a customizable and interactive environment for navigating and browsing complex information spaces. [SBM+02]. The primary view in SHriMP uses a zoom interface to explore hierarchical software structures. The zoom interface provides advanced features to combine a hypertext-browsing metaphor with animated zooming motions over nested graphs. Filtering, abstraction and graph layout algorithms are used to reveal complex structures in the software system under analysis.

SHriMP presents a nested graph view of a software architecture. Program source code and documentation are presented by embedding marked up text fragments within the nodes of the nested graph. Finer connections among these fragments are represented by a network that is navigated using a hypertext link-following metaphor. SHriMP combines this hypertext metaphor with animated panning and zooming motions over the nested graph to provide continuous orientation and contextual cues for the user.

SHriMP employs a fully zoomable interface for exploring software. This interface supports three zooming approaches: geometric, semantic and fisheye zooming. A user browsing a software hierarchy may combine these approaches to magnify nodes of interest. Geometric zooming is the simplest type of zooming. A part of the nested view is simply scaled around a specific point in the view. Geometric zooming causes other information to be elided. Fisheye zooming allows the user to zoom on a particular piece of the software, while preserving contextual information. Information that is of interest appears larger than other information which is reduced in size accordingly.

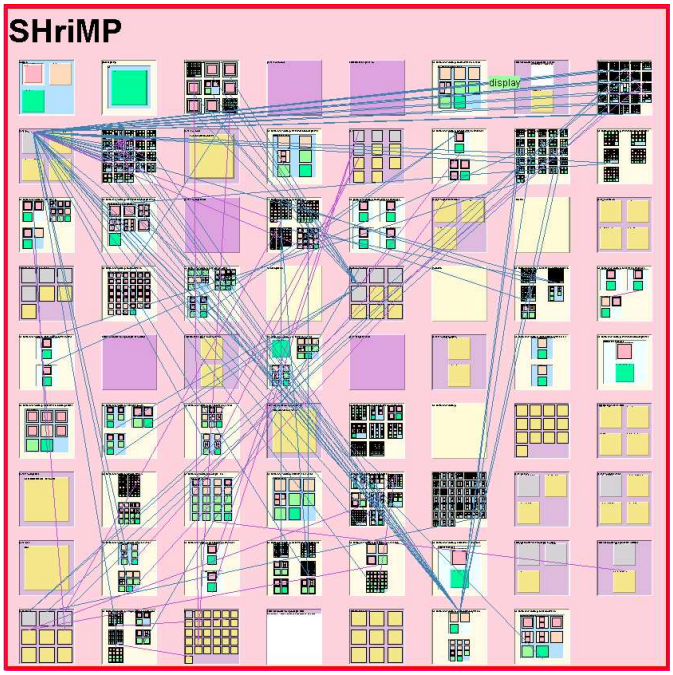

Figure 2.8: A SHriMP view showing the architecture of the SHriMP program itself.

SHriMP also provides a semantic zooming method. When magnified, a selected node will display a particular view depending on the task at hand. For example, when zooming on a node representing a Java package, the node may display its children (packages, classes, and interfaces). Alternatively, it may show its Javadoc, if it exists. Other possible views may include annotation information, code editors or other graphical displays. A node representing a class or interface may display its children (attributes and operations) or it may display the corresponding source code. SHriMP determines which view to show according to the action that initiated the zoom action.

SHriMP is language independent and can be used for browsing any information space. SHriMP supports viewing of parsed C, C++, PL/AS, COBOL and LaTeX code.

2.4 Metrics

Software engineering involves the study of the means of producing high quality software products with predictable costs and schedules. One of the major goals in software engineering is to control the software development process, thereby controlling costs and schedules, as well as the quality of the software products. As a direct result of software engineering research, software metrics have been brought to attention of many software engineers and researches. As De Marco points out, "you cannot control what you cannot measure". Software metrics measure certain aspects of software and can be generally divided into three categories:

- process measurement for understanding, evaluation and improvement of the development method;
- product measurement for the quantification of the product (quality) characteristics and the validation of these measures;
- resource measurement for the estimation of needed resources in terms of human and hardware resources.

Nowadays there is a plethora of metric definitions to evaluate design, code, productivity, and project cost [FP96, HS96]. Metrics have long been studied as a way to assess the quality of large software systems [FP96] and recently this has been applied to object-oriented systems as well [Mar98]. Metrics profit from their scalability and, in the case of simple ones, from their reliable definition. However, simple measurements are hardly enough to sufficiently and reliably assess software quality. Metrics have been also used to identify duplicated code [Kon97]. Some theories exist to assess the well-foundedness of a sound metrics [BDW99].

2.5 Reverse Engineering Environments

Reengineering large industrial software systems is impossible without appropriate tool support. First of all, there is the scalability issue (millions of lines of code) but there is also the extra complexity of supporting and combining multiple tools with a wide variety of tasks (standard forward engineering techniques must be combined with reverse- and re-engineering facilities). The need for tool support in reengineering is reflected by the numerous tools and tool prototypes available in the reengineering research community.

All tool sets have basically a similar structure. There is a repository to store data about software system. There are parsers to extract information from source code and model importers to read in models stored using an exchange format. The tools themselves, browsers, visualizers, etc., use the repository as their information base.

This section discusses existing reengineering tools and tool environments. Several surveys have been compiled [BG97, SS00], but these do not provide exhaustive lists either. There are the general visualizers, not necessarily aimed at software reengineering. Then there are tools that are highly specialized in a certain programming language or even a one vendor-specific dialect. Furthermore, there are the tool environments, which are explicitly aimed at supporting multiple, possibly cooperating tools, and generic meta-data repositories. We focus on the tools that are of interest in the context of modeling software for reengineering.

2.5.1 Moose

MOOSE [NDG05] is an reengineering environment designed to provide the necessary infrastructure for building new tools and for integrating them. Moose centers on a language independent meta-model, and offers services like metrics evaluation, grouping, querying, navigation, and meta-descriptions. Several tools have been built on top of Moose dealing with different aspects of reengineering like: visualization, evolution analysis, semantic analysis, concept analysis or dynamic analysis.

Moose uses a layered architecture. Information is transformed from source code into a source code model. The models are based on the FAMIX language independent meta-model [DTD01]. The information in this model, in the form of entities representing the software artifacts of the target system, can be analyzed, manipulated and used to trigger code transformations by means of refactorings.

Moose supports multiple languages via the FAMIX meta-model. Source code can be imported into the meta-model in two different ways. In the case of VisualWorks Smalltalk the language in which Moose is implemented models can be directly extracted via the meta-model and the parser of the Smalltalk language. For other source languages Moose provides an import interface for CDIF, XMI and MSE files based on the FAMIX meta-model. Over this interface Moose uses external parsers for languages other than Smalltalk. Currently C++, Java, COBOL, and other Smalltalk dialects are supported.

The Moose core contains the FAMIX meta-model, a model repository, an import-export interface, navigation utilities and a set of basic generic tools.

On top of Moose several tools were built, each of them having its own focus: visualizations (CodeCrawler), an information visualization engine (Mondrian), system evolution analysis (Van), dynamic analysis (DynaMoose), semantic analysis (Hapax). These tools extend Moose and collaborate with each other using meta-descriptions.

2.5.2 iPlasma

IPLASMA [MMM+05] is an integrated environment for quality analysis of object-oriented software systems that includes support for all the necessary phases of analysis: from model extraction (including scalable parsing for C++ and Java) up to high-level metrics-based analysis, or detection of code duplication. IPLASMA has three major advantages: extensibility of supported analysis, integration with further analysis tools and scalability, as it was used in the past to analyze large-scale projects in the size of millions of code lines (e.g. Eclipse and Mozilla).

The tool platform, starts directly from the source-code (C++ or Java) and provides the complete support needed for all the phases involved in the analysis process, from parsing the code and building a model up to an easy definition of the desired analyzes including even the detection of code duplication, all integrated by a uniform front-end, namely INSIDER.

Although IPLASMA was developed as a research tool, it is not a "toy". It was successfully applied for analyzing the design of an important number of "real-world"systems including very large-scale systems (> 1 MLOC), like Mozilla (C++, 2.56 million LOC) and eclipse, (Java, 1.36 million LOC).

2.5.3 Rigi

RIGI [SWM97] is a public domain tool developed in the Rigi Research Project at the University of Victoria (Hausi A. Muller). The main component is an editor called *rigiedit*. It is written in RCL, an extended version of Tcl/Tk, and supports viewing of parsed C, C++, PL/AS, COBOL and LaTeX code. A parser for generating the representation used in the Rigi system, called Rigi Standard Format (RSF), is also available. Rigiedit shows the correspondences of the entities that are generated by parsing the application and allows to edit these representations.

The Rigi reverse engineering system provides two solutions for browsing software structures in its graph editor. The first approach uses multiple, overlapping windows, where each window displays a portion of a subsystem hierarchy. A second (newer) approach, the Simple Hierarchial Multi-Perspective (SHriMP) visualization technique, presents software structures using fisheye views of nested graphs.

Chapter 3

Why Lisp?

"Lisp has jokingly been called 'the most intelligent way to misuse a computer'. I think that description is a great compliment because it transmits the full flavor of liberation: it has assisted a number of our most gifted fellow humans in thinking previously impossible thoughts."
Edsger Dijkstra

The first question that arises when reading this paper is "Why Lisp?". Why this old, strange language that nobody uses anymore? This is the general opinion of people when asked about Lisp, but this is wrong. We will show in this chapter why Lisp is a good choice and a language worth studying.

3.1 A Lisp Overview

Lisp is one of the oldest programming languages still in widespread use today (only Fortran is older). Lisp was originally created as a practical mathematical notation for computer programs, based on Alonzo Church's lambda calculus. It quickly became the favored programming language for artificial intelligence research. As one of the earliest programming languages, Lisp pioneered many ideas in Computer Science, including the if/then/else construct, tree data structures, recursive function calls, dynamic memory allocation, garbage collection, first-class functions, lexical closures, interactive programming, incremental compilation, meta-programming and dynamic typing.

3.1.1 Where it all started

Lisp was invented by John McCarthy in 1958 while he was at MIT. McCarthy published its design in a paper in Communications of the ACM in 1960, entitled "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I" [McC60] (Part II was never published). He showed that with a few simple operators and a notation for functions, one can build a Turing-complete language for algorithms.

Lisp was first implemented by Steve Russell on an IBM 704 computer. Russell had read McCarthy's paper, and realized (to McCarthy's surprise) that the *eval* function could be implemented as a Lisp interpreter.

Since its inception, Lisp was closely connected with the artificial intelligence research community. Lisp was the favorite tool for programmers writing software to solve hard problems such as automated theorem proving, planning and scheduling, and computer vision. These were problems that required a lot of hard-to-write software; to make a dent in them, AI programmers needed a powerful language, and they grew Lisp into the language they needed.

Over its almost fifty-year history, Lisp has spawned many variations on the core theme of an S-expression language. Lisp is a family of computer programming languages, with a lot of dialects and with a long history and a distinctive fully-parenthesized syntax. Today, the most widely-known general-purpose Lisp dialects are Common Lisp and Scheme. In 1996, the American National Standards Institute (ANSI) released a standard for Common Lisp (ANSI standard X3.226-1994) that built on and extended the language specified in CLtL2 [Ste90], adding some major new features such as the CLOS and the condition system.

So, on one hand, Lisp is one of Computer Science's "classical" languages, based on ideas that have stood the test of time. On the other, it's a thoroughly modern, general-purpose language whose design reflects a deeply pragmatic approach to solving real problems as efficiently and robustly as possible.

3.1.2 General features

No syntax. Probably the most striking feature of Lisp is that it has no syntax. All program code is written as *S-expressions*, or parenthesized lists. So we could say that Lisp has no syntax. Lisp was the first homoiconic programming language: the primary representation of program code is the same type of list structure that is also used for the main data structures. As a result, Lisp functions can be manipulated, altered or even created within a Lisp program without extensive parsing or manipulation of binary machine code. This is generally considered one of the primary advantages of the language with regards to its expressiveness, and makes the language amenable to metacircular evaluation.

The REPL. Lisp languages are frequently used with an interactive command line, which may be combined with an integrated development environment. The user types in expressions at the command line, or directs the IDE to transmit them to the Lisp system. Lisp "reads" the entered expressions, "evaluates" them, and "prints" the result. For this reason, the Lisp command line is called a "*read-eval-print loop*", or REPL. To implement a Lisp REPL, it is necessary only to implement these three functions and an infinite-loop function. Naturally, the implementation of eval will be complicated, since it must also implement all special operators like *cond*. This done, a basic REPL itself is but a single line of code: *(loop (print (eval (read))))*.

Macros. Common Lisp's macro feature is one of the highlights of this language – it is a very powerful means to write your own programming language constructs beyond mere functions. Contrary to common belief, they are different from C macros. Macros in Lisp provide a very powerful and flexible method of extending Lisp syntax. They are much more powerful than simple string substitution, as in C, Lisp macros are a full-fledged code-generation system. Lisp

macros are Lisp programs that generate other Lisp programs. Although extremely powerful and useful, macros are also significantly harder to design and debug than normal Lisp functions, and are normally considered a topic for the advanced Lisp developer. Lisp macros take Lisp code as input, and return Lisp code. They are executed at compiler pre-processor time, just like in C. The resultant code gets executed at run-time.

CLOS. Common Lisp Object Systems (CLOS) is different from other object-oriented languages, like Java or Smalltalk. CLOS offers full object-orientation with classes, subclassing, multiple inheritance, multi-methods and before-, after- and around advices and off-course the so-called Meta-Object Protocol. CLOS is a multiple dispatch system. This means that methods can be specialized upon the types of all of their arguments. Consequently CLOS methods do not belong to classes. Having multiple dispatch, methods conceptually belong to each class they dispatch on, but they do not syntactically belong to one or another class. Methods in CLOS are grouped into generic functions; a generic function is a collection of methods with the same name and argument structure, but with differently-typed arguments. A generic function is a function whose behavior depends on the classes or identities of the arguments supplied to it. The methods associated with the generic function define the class-specific operations of the generic function. CLOS is dynamic, meaning that not only the contents, but also the structure of its objects can be modified at runtime. CLOS is also reflective, meaning that programs can inspect its own structure and can also modify its structure dynamically.

The Metaobject Protocol. A metaobject protocol (MOP) is an interpreter of the semantics of a program that is open and extensible. Therefore, a MOP determines what a program means and what its behavior is, and it is extensible in that a programmer (or meta-programmer) can alter program behavior by extending parts of the MOP. The MOP exposes some or all internal structure of the interpreter to the programmer. The MOP may manifest as a set of classes and methods that allow a program to inspect the state of the supporting system and alter its behavior. MOPs are implemented as object-oriented programs where all objects are meta-objects.

The the best-known runtime MOP and the most powerful is the one described in the book "*The Art of the Metaobject Protocol*"[KdRB91]; it applies to the Common Lisp Object System (CLOS) and allows full reflection (introspection and intercession) on every entity in CLOS (object, classes, methods, slots) and even on the mechanisms of inheritance, method dispatch, class instantiation, etc. The use of The Metaobject Protocol gives CLOS two unusual proprieties: allows users and external programs to inspect the internals of CLOS environments and allows external programs to extend the CLOS language itself without modifying existing implementation code and without affecting other, existing programs.

A truly multi-paradigm language. Started as a functional language, today Lisp is a truly multi-paradigm programming language. It incorporates all major programming paradigms: functional, procedural, logic, object-oriented and even newer paradigms like aspect-oriented. The secret of Lisp is that it does not have to change each time a new paradigm is introduced, because introducing a new paradigm into Lisp is like adding a new library, the language remains the same. All this flexibility is won because of macros. The best example of Lisp's flexibility and extensibility is CLOS, a full object-system built on top of Lisp with a set of macros. That is the reason why John Foderaro called Lisp "*a programmable programming language*"[Fod91]. And probably this

is the explanation for Lisp's survival for so many years and so many changes. Because, with each new change, Lisp adopted it very quickly and even took the new paradigms further by being a testbed for rapidly testing new ideas. Lisp's flexibility allowed it to adapt as programming styles change, but more importantly, Lisp can adapt to a particular programming problem. In other languages the programmer fits the problem to the language; with Lisp he extends the language to fit the problem. Lisp allows you to add new constructs to the language very easily. If in other languages a new construct is added when a new release of the language is issued, in Lisp every programmer is also a language designer. More than any other language, Lisp follows the philosophy that what's good for the language's designer is good for the language's users. "Not only can you program in Lisp but you can program the language itself."(John Foderaro) [Fod91]

Rapid prototyping. Lisp is particularly good at supporting *rapid prototyping* [Pit94]. Two of the main reasons are the REPL and that Lisp is a dynamic typed language. The fact that Lisp is a dynamic typed language is that you do not waste all your time thinking at type problems and you can focus more on the programming itself. Dynamic languages are very good at prototyping and experiencing new ideas, because the delay between idea and runnable program is much more shorter. Lisp, and dynamic languages in general, do not restrain you from expressing naturally your thoughts directly, without having to do the extra boilerplate code before any work is done. Another advantage of Lisp for rapid prototyping is it's REPL. The REPL is the ideal interface for testing new ideas, where you just write code and evaluate it and get the answer immediately, without having to wait for a long compilation cycle or without any pre-work. The code you write in the REPL becomes a fully integrated part of your system as you write it, and you never have to leave your lisp session to rebuild and start over. The development style ends up reflecting the dynamic powers of the language itself: you can redefine your functions and classes, as often as you want, both in development and in the running application.

Bottom-up programming. Experienced Lisp programmers tend to divide their programs differently [Gra04]. As well as top-down design, they follow a principle which could be called bottom-up design – changing the language to suit the problem. In Lisp, you don't just write your program down toward the language, you also build the language up toward your program. Language and program evolve together, one towards each other. In the end your program will look as if the language had been designed for it. And when language and program fit one another well, you end up with code which is clear, small and efficient. This style of programming is more recently coined as programming with *domain specific languages*. Each "mini-language" you obtain when you use bottom-up programming could be called a domain specific language, where the domain is the one of the problem you are resolving. It's worth emphasizing that bottom-up design doesn't mean just writing the same program in a different order. When you work bottom-up, you usually end up with a different program. Instead of a single, monolithic program, you will get a larger language with more abstract operators and a smaller program written in it. There are several advantages for bottom-up programming: by making the language do more of the work, bottom-up design yields programs which are smaller and more agile; bottom-up design promotes code re-use, any of the utilities you wrote for the first program will also be useful in the succeeding ones; bottom-up design makes programs easier to read; working bottom-up helps to clarify your ideas about the design of your program, because it causes you always to be on the lookout for patterns in your code.

3.1.3 Myths and Legends

Lisp is too old. Most of the people consider Lisp as an outdated language. But this couldn't be more wrong, because even after almost fifty years Lisp still is a modern language with new and innovative ideas that even now aren't found in any other programming language.

Lisp is slow. The reality is that a Lisp compiler can produce native machine code that is comparable in speed with other languages (including C).

Lisp is not compiled. Even if initially Lisp was only interpreted, all modern implementation provide a Lisp compiler and also an interpreter. In fact, much important work in the theory of program compilation has been done using Lisp.

Lisp is not standard. X3.226/1994, the American National Standard for Programming Language Common Lisp, not only exists but in fact was the first ANSI standard for an object-oriented programming language.

Lisp is for AI. Even if Lisp was the primary choice for artificial intelligence programming, this does not mean that Lisp is not used in other areas. As Kent Pitman says: "Please don't assume Lisp is only useful for Animation and Graphics, AI, Bioinformatics, B2B and E-Commerce, Data Mining, EDA/Semiconductor applications, Expert Systems, Finance, Intelligent Agents, Knowledge Management, Mechanical CAD, Modeling and Simulation, Natural Language, Optimization, Research, Risk Analysis, Scheduling, Telecom, and Web Authoring just because these are the only things they happened to list."

Lisp syntax is painful. The truth is that Lisp has no syntax. All program code is written as S-expressions, or parenthesized lists. This approach has a lot of advantages: easy to teach, easy to parse, code and data are the same, text editors can provide better support.

3.2 Why is Lisp Different

Even from its conception, Lisp was different from other languages (mainly Fortran at that time). Even if other programming languages borrowed ideas from Lisp over the years, Lisp is still somehow different. Lisp sets itself apart because it's a dynamic, homoiconic, multi-paradigm, reflective, meta-circular language.

Modeling Lisp systems, as we'll see later, is different from other languages. First of all the main reason is that Lisp is a multi-paradigm language, it combines more programming paradigms and styles in one language. But even in the context of object-oriented programming languages, for example, Lisp sets itself apart from other languages with different language entities and new ways of combining those entities, as well see it later.

From the meta-modeling point of view the main difficulties are Macros and CLOS. Macros are a unique feature that aren't to be found in any other languages (they are very different from C macros). And CLOS is a very different object system from other object-oriented languages, like Java, C++ or even Smalltalk. The next two paragraphs will describe in more detail Macros and CLOS and then Chapter 4 will provide a solution for modeling Lisp systems, based on an existing meta-model.

3.2.1 Macros

Common Lisp's macro feature is one of the highlights of this language - it is a very powerful means to write your own programming language constructs beyond mere functions. Macros in Lisp provide a very powerful and flexible method of extending Lisp syntax. Because language extensibility is a very important property, but also very hard to realize, macros are the ideal way of extending the language in a very portable way. as Guy Steele said: "From now on, a main goal in designing a language should be to plan for growth."[Ste90].

The best application of macros is adding a domain-specific notation to the language, overlaying a little language on the top of a general-purpose one. Paul Graham have makes compelling arguments for such embeddings [Gra93]. Macros give the programmer the power of abstraction where neither functional nor object abstraction will suffice. With macros the programmer can abstract the complexity of a program by creating new language constructs and so hiding a part of the implementation and winning in clarity, concision and code re-use. Combining Lisp macros with bottom-up programming results in a novel style of programming. This way you write your program down toward the language and also build the language up toward your program. Language and program evolve together, and at the end you get a new domain-specific language embedded in your language, that you can reuse for other programs.

Contrary to common belief Lisp macros are very different from C macros. They are much more powerful than simple string substitution, as in C, Lisp macros are a full-fledged code-generation system. Lisp macros are Lisp programs that generate other Lisp programs.

Although extremely powerful and useful, macros are also significantly harder to design and debug than normal Lisp functions, and are normally considered a topic for the advanced Lisp developer. Naturally, every powerful programming construct invites misuse. Misuse of macros may cause the generation of an incorrect program or the triggering of errors during compilation. They are difficult to develop and test. It takes a degree in "macrology" to write even a moderately complex macro and it takes even more advanced degree to understand the macro.

Macros are implemented somewhat differently in Lisp than they are in C. Instead of a separate, preprocessing-based language layer, macros are executed at compile time, interleaved with the normal compilation. When the compiler encounters a parse tree whose head names a macro, the compiler executes the body of the macro, with the other branches of the parse tree bound as the arguments to the macro. The result of executing a macro definition's body is a new parse tree, which the compiler then proceeds to compile as usual.

One thing to notice is that the body of the macro is arbitrary Lisp code. The complete power of the programming language is available to the programmer at compile time for the purpose of expanding the use if a macro. This is very different from most macro systems, and C's in particular, which has only very limited expressiveness available to the macro system (in C the macro system is limited to only string substitution).

Since macros are called and return values, they tend to be associated with functions. Macro definitions sometimes resemble function definitions and programmers consider a macro a "built-in function". And programmers also confuse macros with simple function and often write macros when a function is enough. Macros work differently from normal functions, and knowing how and why macros are different is the key to using them correctly. A function produces results, but a macro produces expressions, which, when evaluated, produce results.

Examples. The best way to understand is to see an example. Suppose we want to write a macro *nil!*, which sets its argument to nil. Paraphrased in English, this definition tells Lisp: "Whenever you see an expression of the form (nil! var), turn it into one of the form (setq var nil) before evaluating it." The expression generated by the macro will be evaluated in place of the original macro call. The step of building the new expression is called macro-expansion. Lisp looks up the definition of nil!, which shows how to construct a replacement for the macro call. The definition of nil! is applied like a function to the expressions given as arguments in the macro call. It returns a list of three elements: setq, the expression given as the argument to the macro, and nil. In this case, the argument to nil! is x, and the macro-expansion is (setq x nil). After macro-expansion comes a second step, evaluation. Lisp evaluates the macro-expansion (setq x nil) as if you had typed that in the first place. Off-course macros can get much more complicated than the example above, and also a macro-expansion can lead to another macro that will also be expanded until no macros remain.

For a more real example of macros we'll take two macros: *DOLIST* and *DOTIMES*.

DOLIST loops across the items of a list, executing the loop body with a variable holding the successive items of the list. This is the basic skeleton (leaving out some of the more complex options):

```
(dolist (var list-form)
     body-form*)
```

When the loop starts, the list-form is evaluated once to produce a list. Then the body of the loop is evaluated once for each item in the list with the variable var holding the value of the item.

DOTIMES is the high-level looping construct for counting loops. The basic template is much the same as DOLISTs.

```
(dotimes (var count-form)
     body-form*)
```

The count-form must evaluate to an integer. Each time through the loop var holds successive integers from 0 to one less than that number.

In previous examples we just defined two new control abstraction, two new iterators. After defining them we can use them as they were part of the language itself.

DOLIST is similar to Perls *foreach* or Pythons *for*. Java added a similar kind of loop construct with the enhanced *for loop* in Java 1.5. A Lisp programmer who notices a common pattern in their code can write a macro to give themselves a source-level abstraction of that pattern. A Java programmer who notices the same pattern has to convince Sun that this particular abstraction is worth adding to the language. Then Sun has to publish a JSR and convene an industry-wide expert group to hash everything out. That process, according to Sun, takes an average of 18 months. After that, the compiler writers all have to go upgrade their compilers to support the new feature. And even once the Java programmers favorite compiler supports the new version of Java, they probably still cant use the new feature until theyre allowed to break source compatibility with older versions of Java. So an annoyance that Common Lisp programmers can resolve for themselves within five minutes plagues Java programmers for years.

Many of the difficulties programmers encounter with macros are because the confusion between macro-expansion and evaluation. Macros are expanded at compile time (they are evaluated only once) and the result is some Lisp code that will be evaluated at run-time.

Many languages offer some form of macro, but Lisp macros are singularly powerful. When a file of Lisp is compiled, a parser reads the source code and sends its output to the compiler. With macros, we can manipulate the program while its in this intermediate form between parser and compiler. Being able to change what the compiler sees is almost like being able to rewrite it. We can add any construct to the language that we can define by transformation into existing constructs.

3.2.2 Common Lisp Object System (CLOS)

The Common Lisp Object System (CLOS) is the facility for object-oriented programming which is part of ANSI Common Lisp. CLOS is a dynamic object system which differs radically from the OOP facilities found in more static languages such as C++ or Java. CLOS was inspired by earlier Lisp object systems such as MIT Flavors and Common LOOPS, although it is more general than either. Originally proposed as an add-on, CLOS was adopted as part of the ANSI standard for Common Lisp.

The Common Lisp Object System is an object-oriented system that is based on the concepts of generic functions, multiple inheritance and method combination [DG87]. All objects in the Object System are instances of classes that form an extension to the Common Lisp type system. The Common Lisp Object System is based on a meta-object protocol that renders it possible to alter the fundamental structure of the Object System itself.

The Common Lisp Object System View of Object-Oriented Programming In the process of adding support for object-oriented programming in Lisp, the designer didn't just copied the existing object-oriented systems, but created a novel object system, which stands out from other object-oriented languages [BGW93]. Following we present some of the design choices that make Lisp's object system so different.

Multiple inheritance. The Common Lisp Object System is a multiple-inheritance system, that is, it allows a class to directly inherit the structure and behavior of two or more otherwise unrelated classes. If no structure is duplicated and no operations are multiply-defined in the several super-classes of a class, multiple inheritance is straightforward. If a class inherits two different operation definitions or structure definitions, it is necessary to provide some means of selecting which ones to use or how to combine them. The Object System uses a linearized class precedence list for determining how structure and behavior are inherited among classes.

Generic functions. The Common Lisp Object System is based on generic functions rather than on message-passing. This choice is made for two reasons: there are some problems with message-passing in operations of more than one argument and the concept of generic functions is a generalization of the concept of ordinary Lisp functions. Generic functions provide not only a more elegant solution to the problem of multiary operations but also a clean generalization of the concept of functions in Common Lisp. Each of the methods of the generic function provides a definition of how to perform an operation on arguments that

are instances of particular classes or of subclasses of those classes. The generic function packages those methods and selects the right method or methods to invoke. In the generic function approach, objects and functions are autonomous entities, and neither is a property of the other. Generic functions decouple objects and operations upon objects; they serve to separate operations and classes. In the case of multiary operations, the operation is a generic function, and a method is defined that performs the operation on the objects (and on objects that are instances of the same classes as those objects).

Method combination. The Common Lisp Object System supports a mechanism called method combination. Method combination is used to define how the methods that are applicable to a set of arguments can be combined to provide the values of a generic function. In many object-oriented systems, the most specific applicable method is invoked, and that method may invoke other, less specific methods. When this happens there is often a combination strategy at work, but that strategy is distributed throughout the methods as local control structure. Method combination brings the notion of a combination strategy to the surface and provides a mechanism for expressing that strategy.

First class objects. In the Common Lisp Object System, generic functions and classes are first-class objects with no intrinsic names. It is possible and useful to create and manipulate anonymous generic functions and classes. The concept of first-class is important in Lisp-like languages. A first-class object is one that can be explicitly made and manipulated; it can be stored anywhere that can hold general objects. Generic functions are first-class objects in the Object System. By making generic functions first-class objects, the designers'goal was to merge the object-oriented paradigm with the function-oriented style of Lisp. Generic functions can be used in the same ways that ordinary functions can be used in Common Lisp.

Reflection. *Reflection* is the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation: introspection and intercession. *Introspection* is the ability for a program to observe and therefore reason about its own state. *Intercession* is the ability for a program to modify its own execution state or alter its own interpretation or meaning. Most of the object-oriented languages, like Java or C#, have only introspectional capabilities, only a few, like Smalltalk, have also intercessional capabilities, but they are limited in power. CLOS has both introspectional and intercessional capabilities. A CLOS program can inspect its own state in detail, but it can also alter that state dynamically. The program can add during run-time a new class, or redefine a class, even if instances of that class exists, or define a new attribute to a class, or define new methods, or off-course redefine existing methods. The ability of reflecting on its own state is owed to the fact most of the CLOS entities are first-class entities (classes, attributes, methods, generic functions and even instances).

The Metaobject Protocol The Metaobject Protocol defines a standard interface to the underpinnings of the CLOS implementation, treating classes themselves as instances of metaclasses, and allows the definition of new metaclasses and the modification of basic class behavior. The flexibility of the CLOS MOP prefigures aspect-oriented programming, which was later developed by some of the same engineers, such as Gregor Kiczales [Pae93]. The meta-object protocol is

designed for use by implementors who need to tailor the Object System for particular applications and by researchers who wish to use the Object System as a prototyping tool and delivery environment for other object-oriented paradigms. The Common Lisp Object System is implemented in terms of a set of objects that correspond to predefined classes of the system. These objects are termed meta-objects. Because meta-objects underlie the rest of the object system, they may be used to define other objects, other ways of manipulating objects, and hence other object-oriented systems.

3.3 Summary

In conclusion, modeling Lisp system using existent meta-models, like the ones described in Chapter 2, is if not impossible, at least incomplete. As we have seen in this chapter, Lisp has some unique features, that aren't to be found in other languages. Consequently, those features aren't included in the existent meta-models, because they're not needed.

From the meta-modeling point of view the main features that can't be modeled with existing meta-models are Macros, with macro-expansions, and CLOS entities, like generic functions and multi-dispatching methods. Including Macros into the meta-model is essential because if information regarding macros would be neglected, a lot of the information that "hides" behind macro-expansions would not be modeled and would not be available to the person who is trying to understand the system, resulting in a incomplete image of the system. Also, including CLOS entities into the meta-model is essential if we want to model CLOS systems (object-oriented Lisp systems). Modeling CLOS systems with any existent object-oriented meta-model is impossible because Lisp's object system, based on the concepts of generic functions, multiple inheritance, multi-dispatch and method combination, is incompatible with the existing C++/Java style object systems.

In Chapter 4 we will provide a solution for modeling Lisp systems, based on an existing meta-model.

Chapter 4

The FAMIX-Lisp Meta-model

"All problems in computer science can be solved by another level of indirection."
Butler Lampson

4.1 Extending the FAMIX Meta-model

In the spirit, of reuse our primary approach was to start from an existing meta-model and extend it to be able to model Lisp systems. The chosen meta-model is FAMIX, presented in Chapter 2. The FAMIX meta-model is a language independent meta-model, which models source code at the program entity level. It also provides an easy way to extend the meta-model with language specific plug-ins.

Why not UML? UML (Unified Modeling Language) is currently embraced as "the"standard in object-oriented modeling languages [SMHP+04]. But UML is not sufficient to serve as a meta-model for reengineering applications because it is forced to rely on UML's built-in extension mechanisms to adequately model the reality in source-code and since UML is specifically targeted towards OOP, it lacks some concepts that are necessary in order to adequately model source-code, in particular the concept of a "method invocation" and an "attribute access" [DDT99]. Of course it is possible to extend UML to incorporate these concepts, but then the protection of the standard is abandoned and with that the reliability necessary to achieve true inter-operability between tools.

The FAMIX meta-model was intended as a language independent meta-model. FAMIX supports multiple hybrid procedural, object-oriented languages, like C++, Java, Smalltalk or Ada. But, as stated by the designers, the aim was not to cover all aspects of all languages, but rather to capture the common features that we need for reengineering activities. As we have seen in Chapter 3, Lisp is different from other mainstream programming languages, making it impossible to model Lisp systems with existing meta-models. So we have chosen to extend the FAMIX meta-model, creating the FAMIX-Lisp meta-model.

Modeling Lisp systems, as we'll see later in this chapter, is different from other languages. One the main reasons is that Lisp is a multi-paradigm language, it combines more programming paradigms and styles in one language. But even in the context of object-oriented programming

languages, for example, Lisp sets itself apart from other languages with different language entities and new ways of combining those entities, as well see it later.

In the process of extending the chosen meta-model, we have identified the main issues in modeling Lisp systems in Chapter 3. From the meta-modeling point of view the main difficulties are Macros and CLOS. Macros are a unique feature that aren't to be found in any other languages (they are very different from C macros) and CLOS is a very different object system from other object-oriented languages, like Java, C++ or even Smalltalk. In the next two sections we will describe the process of extending the FAMIX meta-model for Lisp. The extension is done in two steps: first we add macros to the meta-model and then we add support for CLOS.

One of the main concerns in extending the meta-model is backwards compatibility. We want to extend the model and not to change any existing entities of the model, that would make the model incompatible with the previous version. So we want that the resulting meta-model remains a language independent meta-model that can model the previous supported languages and in addition can also model Lisp systems. Consequently our goal is to extend the meta-model only with a minimal number of new entities and attributes, without any modification to the existing entities.

For a preview of the magnitude of the extension we present in Figure 4.1 the initial FAMIX meta-model in the left and the new extended FAMIX-Lisp meta-model in the right with all the extensions colored in red (this figure is not intended to be read, but only for a general view).

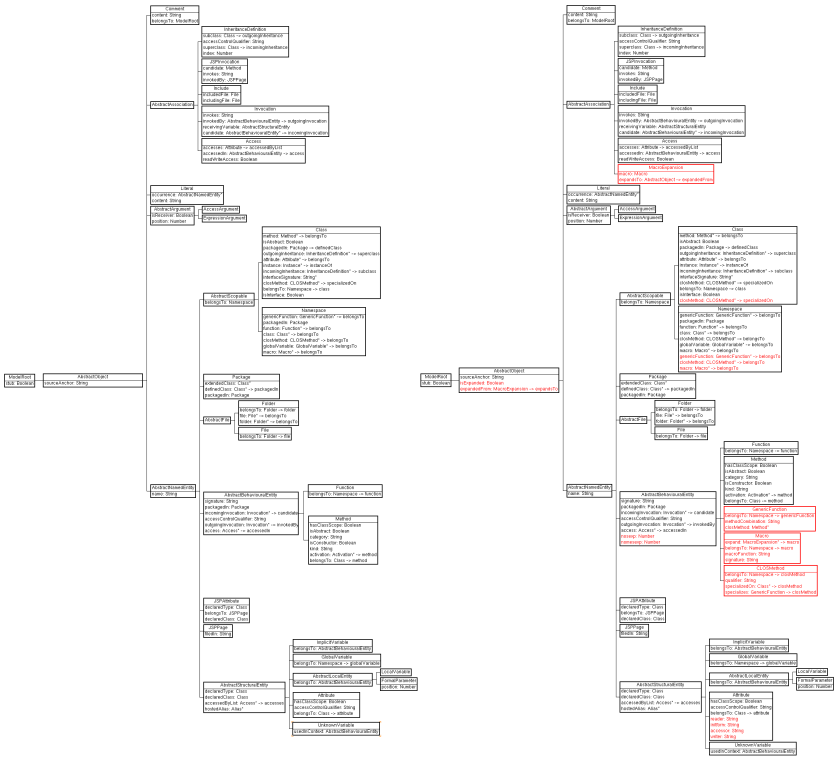

Figure 4.1: The FAMIX Meta-model: before and after

4.1.1 Adding Macros

Up until recently, information on the preprocessor directives has typically not been made part of the meta-models used in program understanding. This is understandable as the C/C++ preprocessor directives form a separate language which cannot be expressed with the model entities of the language. There is some work on integrating preprocessor information in the meta-models used in program understanding [LS94, EBN02], but all work focused only on the C/C++ preprocessor, the only macro system from a mainstream programming language.

Including Macros into the meta-model is essential because if information regarding macros would be neglected, a lot of the information that "hides" behind macro-expansions would not be modeled and would not be available to the person who is trying to understand the system, resulting in a incomplete image of the system. This is a bigger issue for Lisp systems than for C/C++, because if the C/C++ preprocessor is limited in power to only some pattern based string substitution, Lisp macros form a fully-fledged code-generation system (Lisp code that generates Lisp code). There are Lisp programs that are making heavily use of macros and if macros would be neglected, the understanding of the program would be very shallow, because the extracted model of the program is incomplete and do not reveal all the information regarding the modeled system.

The best example of macro usage is probably the Common Lisp Object System, which is a set of complex macros, that hide all the implementation details of the object system. Imagine that if CLOS were'nt a language extension, but only part of an application, how much information would be lost in the process of model extraction from such a system, if macro would'nt be supported.

This section introduces new FAMIX entities that model information about the macro system of Lisp. To respect our goal of a minimal extension to the meta-model and because of Lisp's uniformity of runtime code and macro code, macro expansions are in fact only execution of normal Lisp code, we managed to model the Lisp's macro system with only two new entities: *Macro* and *MacroExpansion*.

Introducing new FAMIX entities: *Macro*. A macro definition has a name, the name of the macro, a parameter list, and a body, the expansion code, the code that will be executed to obtain the code that will substitute the macro call.

We introduce the new Macro entity to model each macro definition from the program. The newly created class will inherit from AbstractBehaviouralEntity (see Figure 4.2 for a overview of the abstract part of the meta-model). The Macro class with its attributes is presented in Table 4.1.

Table 4.1: The Macro entity

Macro
name: String
signature: String
macroFunction: String
belongsTo: Namespace
expand: MacroExpansion

Attributes of the MACRO class:

- *name* is inherited from ABSTARCTNAMEDENTITY, which is a super-class of ABSTRACTBEHAVIOURALENTITY;
- *signature* is a string containing the parameter list of the macro;
- *macroFucntion* is the body of the macro, also called macro function, stored as a string;
- *belongsTo* is the namespace in which the macro was defined;
- *expand* is a list of macro-expansions generated by the macro (see Table 4.2 for the definition of MACROEXPANSION)

Introducing new FAMIX entities: *MacroExpansion*. The process of macro expansion is done at compile-time. The compiler scans the source code for macros and when it encounters one it will execute the body of the macro, the attached macro function, with the parameters provided in the source code, and then it will substitute the macro call with the result of the expansion. When Lisp is interpreted, rather than compiled, the distinction between macro expansion time and runtime is less clear because they are temporally intertwined.

In the process of macro expansion, a simple macro call can be expanded to ordinary Lisp code that will contain new program entities, calls or invocations. So we'll model also this "hidden" entities in the process of model extraction, mentioning that they were obtained from a macro expansion.

We introduce the new MACROEXPANSION entity to model the process of macro expansion. The new entity will define the relation between the macro that was expanded and the new entity that was obtained from the process of expansion. The newly created class will inherit from ABSTRACTASSOCIATION (see Figure 4.2 for a overview of the abstract part of the meta-model). The MACROEXPANSION class with its attributes is presented in Table 4.2.

Table 4.2: The MacroExpansion entity

MacroExpansion
macro: Macro expandsTo: AbstractObject

Attributes of the MACROEXPANSION class:

- *macro* is the macro that was expanded;
- *expandsTo* is the entity that was obtained from the process of macro expansion.

Adding new attributes to existing FAMIX entities. Besides creating new entities we also have to add new attribute to existing entities.

In the process of macro expansions we discover new entities that will be included into the model. So we have to distinguish between program entities and macro expanded entities. This is why we add two new attributes to the ABSTRACTOBJECT class (see Figure 4.2 for a overview of

the abstract part of the meta-model). AbstractObject is the root of the FAMIX hierarchy. We add these new attributes to the root of the hierarchy because in the process of macro expansion we can obtain any kind of Lisp entity, because a macro call is expanded to normal Lisp code. The AbstractObject class with its new attributes is presented in Table 4.3.

Table 4.3: The AbstractObject entity

AbstarctObject	
	sourceAnchor: String
NEW	isExpanded: Boolean
NEW	expandsFrom: MacroExpansion

New attributes of the AbstractObject class:

- *isExpanded* shows if the entity is a regular program entity or it is a macro expanded entity;
- *expandsFrom* is the macro that was expanded into this new entity.

4.1.2 Adding support for CLOS

When talking about object-oriented programming languages most people think of C++ and Java, the two most used object-oriented languages today. This is the reason why most of the reverse engineering environment focus only on this mainstream languages. There are little exceptions, like MOOSE that supports Smalltalk, being also developed in Smalltalk. All these meta-models are build on the C++/Java style object systems.

The Common Lisp Object System (CLOS) is an object-system that is based on the concepts of generic functions, multiple inheritance, multi-dispatch and method combination. Consequently CLOS will not be compatible with the existing object-oriented meta-models. This is the reason for creating an extension to the FAMIX meta-model to support the CLOS systems.

This section introduces the new FAMIX entities that model Common Lisp's Object System. For a minimal extension to the meta-model, we added only two new entities: *GenericFunction* and *CLOSMethod*.

Introducing new FAMIX entities: *GenericFunction*. CLOS is based on generic functions rather then on message-passing. A generic function is a collection of methods with the same name and argument structure, but with differently-typed arguments. A generic function is a function whose behavior depends on the classes or identities of the arguments supplied to it. The methods associated with the generic function define the class-specific operations of the generic function. Generic functions decouple objects and operations upon objects; they serve to separate operations from classes.

A generic function defines an abstract operation, specifying its name and a parameter list, but no implementation. A generic function is generic in the sense that it can accept any objects as arguments. However, by itself, a generic function cant actually do anything. The actual implementation of a generic function is provided by methods. When a generic function is invoked, it compares the actual arguments it was passed with the specializers of each of its methods to find the applicable methods, those methods whose specializers are compatible with the actual

arguments. In simple cases, only one method will be applicable, and it will handle the invocation. In more complex cases, there may be multiple methods that apply. They are then combined into a single effective method that handles the invocation, through a process called method combination. The process of method combination is also another powerful mechanism, but a detailed presentation of it is not in the scope of this work.

We introduce the new GenericFunction entity to model each generic function from the program. The newly created class will inherit from AbstractBehaviouralEntity (see Figure 4.2 for a overview of the abstract part of the meta-model). The GenericFunction class with its attributes is presented in Table 4.4.

Table 4.4: The GenericFunction entity

GenericFunction
name: String
signature: String
belongsTo: Namespace
methodCombination: String
closMethod: Method*

Attributes of the GenericFunction class:

- *name* is inherited from AbstarctNamedEntity, which is a super-class of AbstractBehaviouralEntity;
- *signature* is a string containing the name and the parameter list of the generic function;
- *belongsTo* is the namespace in which the generic function was defined;
- *methodCombination* is the method combination applied when there are more than one method applicable to a method call. If nil then we have the standard method combination;
- *closMethod* is a list off all the methods that provide the implementation of the generic function (methods with the same name and congruent argument list with the generic function);

Introducing new FAMIX entities: *CLOSMethod*. CLOS has a multiple dispatch system. This means that methods can be specialized upon the types of all of their arguments. Consequently CLOS methods do not belong to classes. Having multiple dispatch, methods conceptually belong to each class they dispatch on, but they do not syntactically belong to one or another class. Methods in CLOS are grouped into generic functions.

The definition of a method is almost the same as the definition of a function: it has name, a parameter list and a body. The only difference is that the required parameters can be specialized by replacing the parameter name with a two-element list. The first element is the name of the parameter, and the second element is the specializer, the class of the parameter.

The FAMIX meta-model already has a *Method* entity, which refers to methods as defined in C++/Java type object systems. But CLOS type methods are different. First of all, in CLOS methods do not belong to a class, but are independent from classes. The relation between classes and CLOS type methods is the dispatch process. Each method has a parameter list, with some

of the parameters being specialized on classes. Unlike other object systems, CLOS has multiple dispatch, meaning that a method can explicitly specialize more than one of the parameters. They are also called *multi-methods*. This are the reasons we have to introduce a new entity that describes the CLOS type of method.

We introduce the new CLOSMethod entity to model each generic function from the program. The newly created class will inherit from AbstractBehaviouralEntity (see Figure 4.2 for a overview of the abstract part of the meta-model). The CLOSMethod class with its attributes is presented in Table 4.5.

Table 4.5: The CLOSMethod entity

CLOSMethod
name: String
signature: String
belongsTo: Namespace
qualifier: String
specializedOn: Class*
specializes: GenericFunction

Attributes of the CLOSMethod class:

- *name* is inherited from AbstarctNamedEntity, which is a super-class of AbstractBehaviouralEntity;
- *signature* is a string containing the name and the parameter list of the method;
- *belongsTo* is the namespace in which the method was defined;
- *qualifier* is the method's qualifier. In CLOS a method can be *primary* (default), *before*, *after* or *around*. Based on this qualifier CLOS combines methods, when there are more than one applicable method;
- *specializedOn* is a list of the classes that the parameters from the parameter list of the method specialize on;
- *specializes* is the generic function associated with the method.

Adding new attributes to existing FAMIX entities. Besides creating new entities for supporting CLOS modeling, we also have to add new attribute to existing entities, like *Class* and *Attribute*.

As we said earlier, CLOS methods do not belong to classes. They only dispatch on classes. Because we introduced a new type of method into the meta-model, we'll also have to relate it to the *Class* entity. The Class class with its new attribute is presented in Table 4.6.

The new attribute of the Class class:

- *closMethods* is a list of all the methods that dispatch on this class. That is all the methods that have a parameter of this class in the parameter list.

Table 4.6: The Class entity

Class	
	isAbstract: Boolean
	isInterface: Boolean
	attribute: Attribute
	belongsTo: Namespace
	method: Method
NEW	closMethod: CLOSMethod

In CLOS attributes are callled *slots*. Besides the proprieties of a attribute from C++/Java style object systems, CLOS slots can have some extra proprieties, like a initial form (a piece of Lisp code that will be evaluated and the result will be the initial value of the slot), reader, writer or accessor methods (automatically created methods, analog with getter/setter methods from Java terminology). The ATTRIBUTE class with its new attribute is presented in Table 4.7.

Table 4.7: The Attribute entity

Attribute	
	hasClassScope: Boolean
	accesControlQualifier: String
	belongsTo: Class
NEW	initform: String
NEW	reader: String
NEW	writer: String
NEW	accessor: String

New attributes of the ATTRIBUTE class:

- *initform* is the code that will determine the initial value of the slot;
- *reader* is the name of the reader method;
- *writer* is the name of the writer method;
- *accessor* is the name of the accessor method (reader and writer).

The FAMIX Namespace Class. In FAMIX there are two entities determining scope: FAMIX.Package and FAMIX.Namespace. Because MOOSE, the primary implementation of the FAMIX meta-model, was implemented in Smalltalk, FAMIX took the terminology from Smalltalk for *package* and *namespace*. In Smalltalk a package is a only a unit for grouping source code files; it is only an external language concept. And a namespace is a language entity controlling the scope of language entities. So, in this terminology, Lisp packages are more like Smalltalk namespaces, even if their name would suggest pairing them with Smalltalk packages. That is why for each Lisp package we'll have a FAMIX.Namespace associated with it, and not a FAMIX.Package.

Because we introduces new entities into the model, we also had to add some new attributes to the Namespace class, which represents the scope of that entities. The NAMESPACE class with its new attributes is presented in Table 4.8.

Table 4.8: The Namespace entity

Namespace	
	function: Function
	globalVariable: GlobalVariable
	class: Class
	method: Method
NEW	macro: Macro
NEW	genericFunction: GenericFunction
NEW	closMethod: CLOSMethod

New attributes of the ATTRIBUTE class:

- *macro* is a list containing all the macros defined in this namespace;
- *genericFunction* is a list containing all the generic functions defined in this namespace;
- *closMethod* is a list containing all the CLOS type of methods defined in this namespace.

4.2 The FAMIX-Lisp Meta-model

This section gives an overview of the FAMIX-Lisp meta-model. This is the result of extending the FAMIX meta-model to model Lisp systems. In Figure 4.3 we have the full FAMIX-Lisp meta-model, with the Lisp extension colored in red.

The abstract part of the complete model is shown in Figure 4.2. ABSTRACTOBJECT is the root of the model hierarchy. Than we have two types of objects. ABSTRACTENTITY is the class of all the language entities. And ABSTRACTASSOCIATION is the class of the associations and relation between entities, like invocations or accesses. Entities are further derived in three

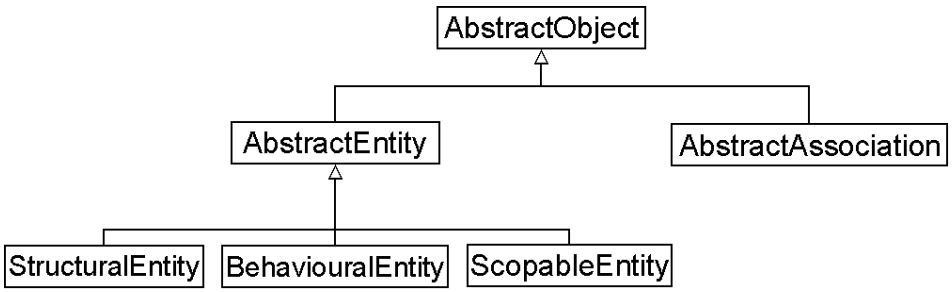

Figure 4.2: The FAMIX-Lisp Meta-model: Abstract part

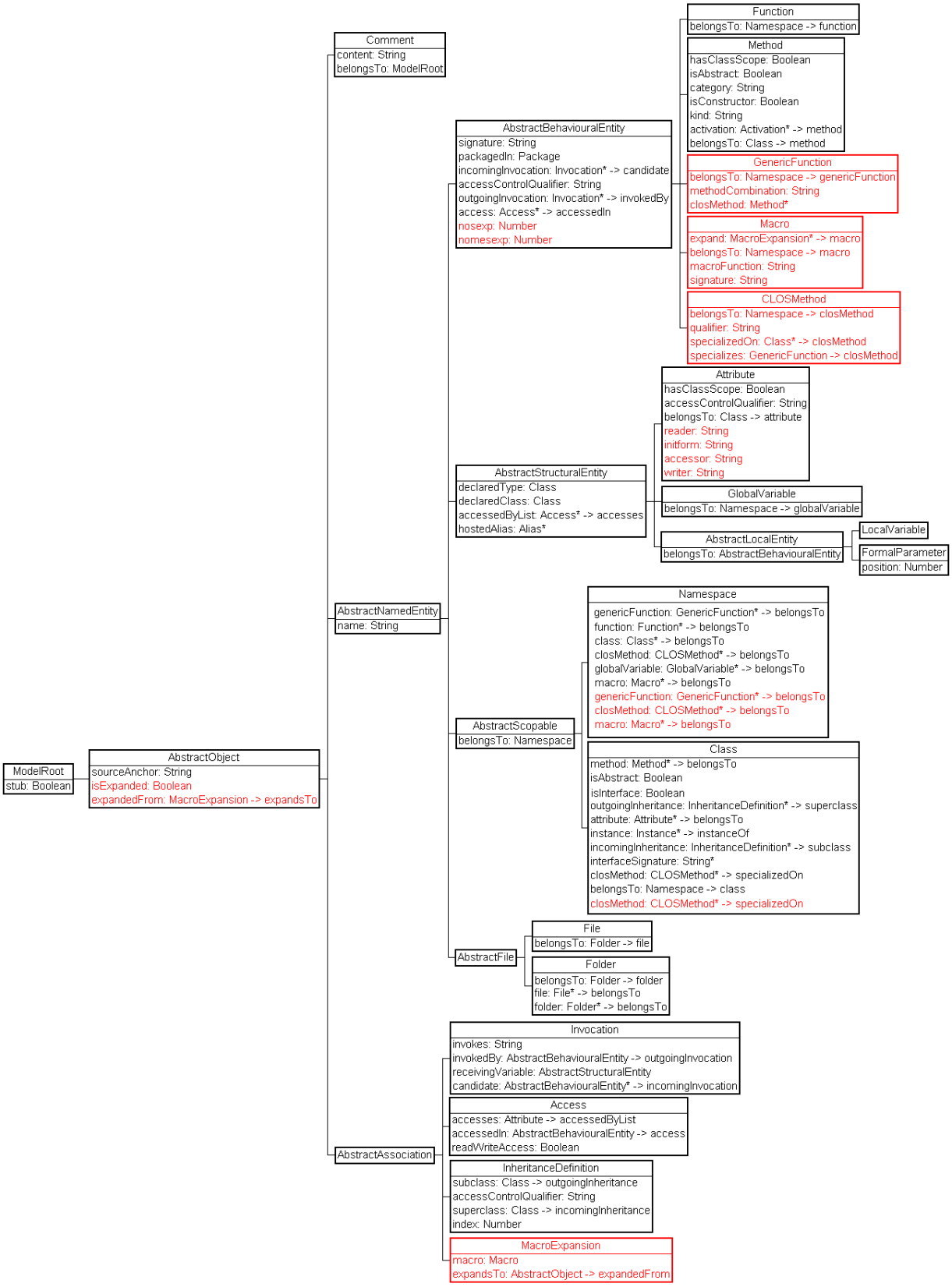

Figure 4.3: The FAMIX-Lisp Meta-model

subclasses. BehavioralEntity is the class of language entities that present behavior, like functions, methods or macros. StructuralEntity is the class of language entities that represent, like global or local variables, class attributes. ScopableEntity is the class of language entities that define a scope in the source code, like namespaces or classes.

Lisp started as a functional programming language and then over the years incorporated most of the programming paradigms: imperative programming, object-oriented, etc. Initially Lisp's object systems were only language libraries and only after standardization it became a language extension. That is why the whole language, even if it has a overall unitary design, it can be at least conceptually separated in sub-parts. The same way in the FAMIX-Lisp meta-model, even if it has a overall cohesive design, we can identify two loose coupled parts of the meta-model: the Lisp part (without the object system) and the CLOS part. We will present each part in the following sub-sections.

4.2.1 The Lisp Core

The core of the functional part of Lisp is shown in Figure 4.4, with the Lisp extension colored in red. The core of the functional part of the meta-model comprises the main entities of the language – namely functions, macros and variables – plus the necessary association between them – accesses, invocations and macro expansions.

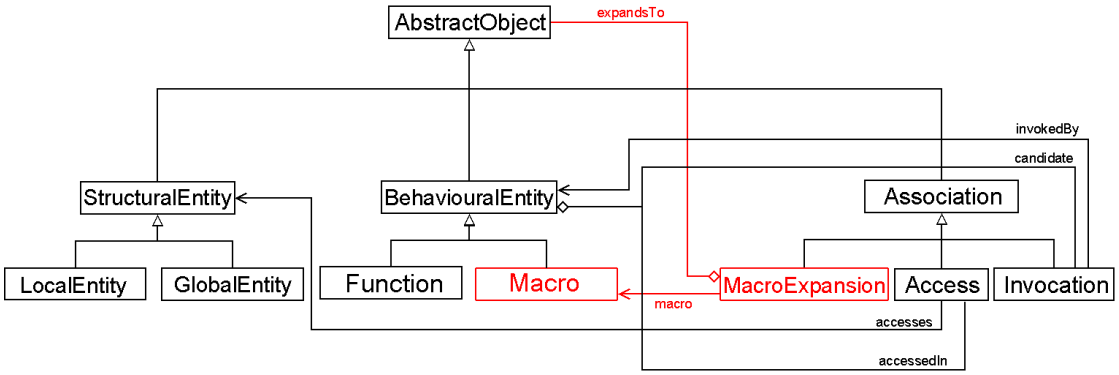

Figure 4.4: The FAMIX-Lisp Meta-model: the Lisp Core

AbstractObject is the root of the model hierarchy. Then we have the StructuralEntity class representing local and global variables and also formal parameters and the BehavioralEntity class representing entities that present behavior, like functions or macro and the Association class representing accesses, invocations or macro expansions. Each access points to the accessed StructuralEntity and to the BehavioralEntity in which it was accessed. Also each invocation points to the invoking BehavioralEntity and to the invoked BehavioralEntity. And each macro expansion points to the Macro that was expanded, the *macro* relation, and to the AbstractObject that was obtained from the expansion, the *expandsTo* relation.

4.2.2 The CLOS Core

The core of the object-oriented part of Lisp, the CLOS part, is shown in Figure 4.5, with the Lisp extension colored in red. The simplified view of the object-oriented part of the meta-model comprises the main object-oriented concepts – namely class, method, attribute and inheritance – plus the necessary associations between them – namely method invocation and attribute access.

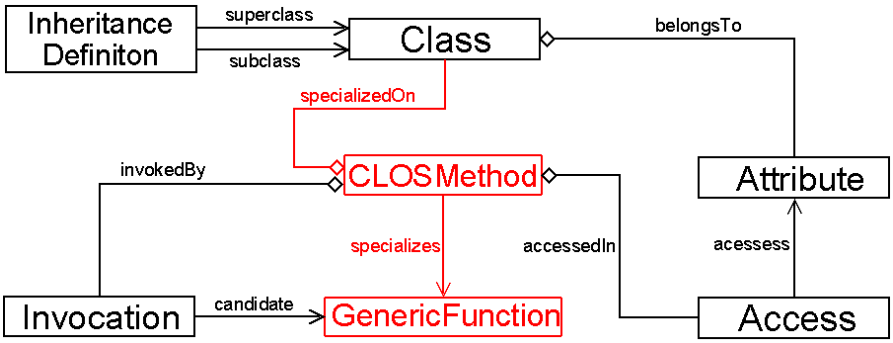

Figure 4.5: The FAMIX-Lisp Meta-model: the CLOS Core

In the center of the CLOS core of the meta-model we have the two new entities, colored in red: CLOSMethod and GenericFunction. Each CLOS method has an associated generic function, pointed by the *specializes* relation. CLOS methods do not belong to a class, they are *specializedOn* a set of classes. Off-course each class can have a superclass or subclass, defined by the InheritanceDefinition, and a set of attributes. Each access points to the accessed Attribute and to the CLOSMethod in which it was accessed. Also each invocation points to the invoking CLOSMethod and to the invoked GenericFunction. We observe here that the invoker and invoked classes are not the same. The invoker class is CLOSMethod because the method contains the implementation of a generic function and the invoked class is GenericFunction because the generic function defines a generic operation and the exact method that will be called can only be known at runtime, Lisp being a dynamic language.

4.3 Examples

For a better understanding of our FAMIX-Lisp meta-model, we'll present some short examples of models. Because presenting a program and the corresponding model would be to large and hard to explain, we'll take another approach. We'll take short parts of a program, a class definition or a function definition for example, and show the corresponding model entities for that piece of code and explain them.

In FAMIX, each model element can be referenced in a unique way. Every model element has a unique identifier, different for each element in a model. So when we have a reference from a entity's property to another entity we'll use that unique identifier to reference it.

The examples presented below will use the MSE format, used by the MOOSE environment. The MSE format allows to specify models for import and export. Similar to XML, MSE is generic and can specify any kind of data, regardless of the meta-model. The MSE format resembles XML,

except for the fact that it uses parenthesis instead of opening and closing tags, similar with Lisp code. As in XML, MSE also has only one root element. That root element has element nodes. Each element node can also have other element nodes, recursively. Each element node has a name and id and any number of attributes nodes. An attribute node can store a value or an reference to another element node.

Lets start with the most basic Lisp entity: a function. We'll present the source code in the left column and the resulting model entities in the right column.

```
(in-package :example)

(defun hello-world ()
    (format t "hello, world"))
```

```
(FAMIX.Namespace (id: 1))
    (name 'example'))
(FAMIX.Function (id: 2)
    (name 'hello-world')
    (belongsTo (idref: 1)))
```

This is the simplest example, we have a function called *hello-world* that only prints a string. The corresponding entity is a FAMIX.Function instance with two properties: the name of the function and the package in which it was defined.

Now let's take a little bit more complex function.

```
(defun print-primes (n)
    (loop for i from 2 to n
        do
        (when (prime i)
            (format t "~D " i))))
```

```
(FAMIX.Function (id: 3)
    (name 'print-prime')
    (belongsTo (idref: 1)))
(FAMIX.FormalParameter (id: 4)
    (name 'n')
    (position 0)
    (belongsTo (idref: 3)))
(FAMIX.Access (id: 5)
    (accesses (idref: 4))
    (accessedIn (idref: 3))
    (readWriteAccess false))
(FAMIX.Invocation (id: 6)
    (invokedBy (idref: 3))
    (candidate (idref: 7)))
...
(FAMIX.Function (id: 8)
    (name 'prime')
    (belongsTo (idref: 1)))
```

We have a function that prints all the prime numbers from 0 to n. For determining if a number is prime it calls a second function, named *prime*. First we have a FAMIX.Function entity corresponding to the *print-prime* function. Than we have a FAMIX.FormalParameter corresponding to the parameter of the *print-prime* function, which has three proprieties: the name of the parameter, the position in the parameter list and the function it belongs to. We also have an access to the n parameter inside the function, which is modeled by the FAMIX.Access entity with three proprieties: the accessed entity, the entity that accessed and if it was a read or

write access. Than we have an call to the *prime* function which determines the FAMIX.Invocation entity, which has two proprieties: the invoker and the invoked entity. Because we needed a reference to the *prime* function we also included the associated entity into the presented model, even if it is not a extracted from the above code, but from the definition of the function.

Now let's take an example involving a macro.

```
(defmacro nil! (x)
    `(setq x nil))

(defun test (v)
    (nil! v))
```

```
(FAMIX.Macro (id: 10)
    (name 'nil!')
    (signature '(n)')
    (belongsTo (idref: 1)))
(FAMIX.FormalParameter (id: 11)
    (name 'x')
    (position 0)
    (belongsTo (idref: 10)))
(FAMIX.Function (id: 12)
    (name 'test')
    (belongsTo (idref: 1)))
(FAMIX.FormalParameter (id: 13)
    (name 'v')
    (position 0)
    (belongsTo (idref: 12)))
(FAMIX.Invocation (id: 14)
    (invokedBy (idref: 12))
    (candidate (idref: 10)))
(FAMIX.Access (id: 15)
    (isExpanded true)
    (expandedFrom (idref: 10))
    (accesses (idref: 13))
    (accessedIn (idref: 12))
    (readWriteAccess true))
(FAMIX.MacroExpansion (id: 16)
    (macro (idref: 10))
    (expandsTo (idref: 15)))
```

We have a macro *nil!* that expands to a attribution, which sets its parameter to nil. Consequently we have a FAMIX.Macro entity and a FAMIX.FormalParamater. To test the macro we added a function that only calls that macro, so we have a FAMIX.Function enity for the function, a FAMIX.FormalParameter for the functions parameter and a FAMIX.Invocation for the macro's invocation within the function. When the macro will be expanded we'll have a new access to the formal parameter *v*. So we have a FAMIX.Access that has the *isExpanded* property set on true and the *expandedFrom* property indicating the macro. We also have a FAMIX.MacroExpansion entity that has two properties: the macro that was expanded and the entity that it expanded to.

Now let's take a look at the object-oriented part of the model. First we'll take a class and a method for example.

```
(in-package :clos-example)

(defclass Person ()
    ((name
      :reader "name"
      :initform "John Doe")))

(defmethod print-name ((p Person))
    (format t "~A" (person p)))
```

```
(FAMIX.Namespace (id: 1))
    (name 'clos-example'))
(FAMIX.Class (id: 2)
    (name 'PERSON')
    (belongsTo (idref: 1)))
(FAMIX.Attribute (id: 3)
    (name 'NAME')
    (hasClassScope false)
    (initform 'John Doe')
    (reader 'NAME')
    (belongsTo (idref: 2)))
(FAMIX.GenericFunction (id: 4)
    (name 'NAME')
    (belongsTo (idref: 1))
    (signature '(NAME (VAR))'))
(FAMIX.Method (id: 5)
    (name 'NAME')
    (signature '(NAME ((VAR PERSON)))')
    (specializedOn (idref: 2))
    (specializes (idref: 4)))
(FAMIX.GenericFunction (id: 6)
    (name 'PRINT-NAME')
    (belongsTo (idref: 1))
    (signature '(PRINT-NAME (P))'))
(FAMIX.Method (id: 7)
    (name 'PRINT-NAME')
    (signature '(PRINT-NAME ((P PERSON)))')
    (specializedOn (idref: 2))
    (specializes (idref: 6)))
(FAMIX.FormalParameter (id: 8)
    (name 'P')
    (position 0)
    (belongsTo (idref: 7)))
(FAMIX.Access (id: 9)
    (accesses (idref: 8))
    (accessedIn (idref: 7))
    (readWriteAccess false)
```

We have a class called *Person*. The corresponding entity is a FAMIX.Class entity with two properties: the name of the class and the namespace in which it was defined. Than we also have a FAMIX.Attribute entity, corresponding to the *name* attribute, with five properties: the name of the attribute, the scope (if it is a instance attribute or a class attribute), it's initial value, the class it belongs to and the reader function. Because the attribute has a reader, we also

have a generic function and a method that CLOS automatically creates for the reader. We also have a simple method called *print-name*. Associated with it we have a FAMIX.Method entity and a FAMIX.GenericFunction for the generic function that CLOS automatically generates for a method definition, if it does'nt exists. The FAMIX.GenericFunction entity has three properties: the name of the function, the namespace it belongs to and the signature. The FAMIX.Method entity has four properties: the name of the method, the signature, the list of classes it specializes on (the classes of the parameters) and the generic functions it implements.

Now let's take a second class that inherits from the first class.

```
(defclass Student (person)
    ((faculty
      :accessor "faculty")))
```

```
(FAMIX.Class (id: 10)
    (name 'Student')
    (belongsTo (idref: 1)))
(FAMIX.InheritanceDefinition (id: 11)
    (subclass (idref: 10))
    (superclass (idref: 2))
    (index 0))
(FAMIX.Attribute (id: 12)
    (name 'FACULTY')
    (hasClassScope false)
    (belongsTo (idref: 10)))
(FAMIX.GenericFunction (id: 13)
    (name 'FACULTY')
    (belongsTo (idref: 1))
    (signature '(FACULTY (VAR))'))
(FAMIX.Method (id: 14)
    (name 'FACULTY')
    (signature '(FACULTY ((VAR STUDENT)))')
    (specializedOn (idref: 10))
    (specializes (idref: 13)))
(FAMIX.GenericFunction (id: 14)
    (name '(SETF FACULTY)')
    (belongsTo (idref: 1))
    (signature '((SETF FACULTY) (VAR))'))
(FAMIX.Method (id: 15)
    (name '(SETF FACULTY)')
    (signature '((SETF FACULTY) ((VAR STUDENT)))')
    (specializedOn (idref: 10))
    (specializes (idref: 14)))
```

We have a *Student* class that inherits from the *Person* class. So besides the FAMIX.Class and the FAMIX.Attribute entities we also have a FAMIX.InheritanceDeifnition, which has three properties: the superclass, the subclass and the position of the superclass in the subclass' inheritance list. Because we have declared an accessor for the *faculty* attribute CLOS automatically declared reader and writer methods and corresponding generic functions. Writer functions, called *setf-functions*, are a little bit stranger because their name is a list of two words (setf and the actual name).

Chapter 5

Visualization Techniques for Lisp Systems

"The commonality between science and art is in trying to see profoundly – to develop strategies of seeing and showing."
Edward Tufte

5.1 Introduction

Visualizations are an important aid for data and also program analysis. By presenting information in a graphical way we make use of the massively parallel architecture of the human visual system for interpreting the data. Software visualization has become one of the major approaches in reverse engineering. Price et al. have presented an extensive taxonomy of software visualization, with several examples and tools [PBS93].

The main target of today's reverse engineering techniques are mainly procedural and object-oriented languages, because they make most of the body of existing legacy systems. Even if object-oriented programming was expected to solve the problem of managing large systems, it has but aggravated this problem, since reading object-oriented code is more difficult than reading procedural code, caused by the difficulties introduced by the technical aspects of object-oriented languages, such as inheritance and polymorphism and the fact that the reading order of a class source code is not relevant as it was in most of the procedural languages where the order of the procedures was important and the use of forward declarations required. So understanding software written in different languages with different programming paradigms has various challenges. For each style of programming there were developed different reverse engineering techniques. But understanding software written in a multi-paradigm language is even harder, because different styles of programming are intertwined in the code. Combining most of the major programming paradigms and having some unique feature, not found in any other programming language, as we've seen in Chapter 3, Lisp rises a bigger challenge for reverse engineering. This is why Lisp needs new techniques and new methods for analyzing and understanding Lisp software.

All our visualizations were developed using Mondrian [MGL06], a visualization framework that supports on-the-fly prototyping of visualizations. Mondrian is a visualization model that is

designed to minimize the time-to-solution. This is achieved by working directly on the underlying data, by making nesting an integral part of the model and by defining a powerful scripting language that can be used to define visualizations. Mondrian supports exploring data in an interactive way by providing hooks for various events. Users can register actions for these events in the visualization script. One of Mondrian's main advanteges is that it can accommodate data provided by third party tools, providing a simple interface through which the programmer can easily script, in a declarative fashion, the visualization. The essence of Mondrian's approach is simple: a script that creates a view, adds nodes representing some objects and adds the edges representing some other objects. The nodes and edges are represented using different shapes.

Our visualizations are interactive, the user can not only see but also interact (zoom, select, move, inspect, etc.) with the views. By making such interactions possible, the gap between the software and the reverse engineers mental model of the software can be further narrowed. Our software visualizations are not a stand-alone solution for reverse engineering Lisp systems, but they are aimed at supporting and complementing other techniques.

In the next sections we'll first apply some of the existing visualizations on Lisp systems and draw the conclusion and then we propose a set of novel visualization for Lisp systems, developed to underline the differences of the language and to help understand and browse Lisp systems.

5.2 Polymetric views and Lisp

In this section we will apply the visualization presented in Section 2.3.1 on models of Lisp systems and analyze the results.

The system complexity view.

We applied the complexity view on some object-oriented Lisp systems and we present here the results.

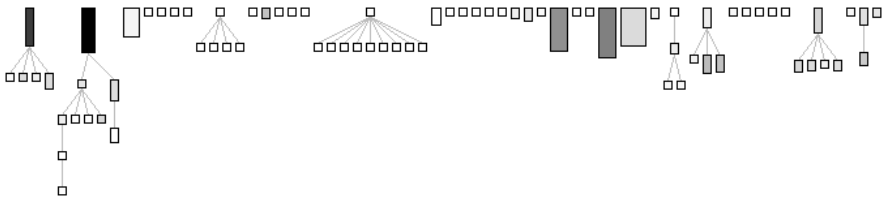

Figure 5.1: Complexity view of a single-inheritance Lisp system

For single-inheritance object-oriented systems, the system complexity view is the same as for any Java or Smalltalk system. The complexity view of single-inheritance Lisp system is shown in Figure 5.1.

For multi-inheritance Lisp systems, the complexity view becomes chaotic. Because of the multi-inheritance system, the view is traversed by a lot of lines and we don't have any more an inheritance tree but more like an graph structure, which has an unordered appearance. We can still spot problem classes, by size or color. But the structure and appearance of the hierarchy is almost useless. This is because of the layout of the view, the tree layout. The layout was

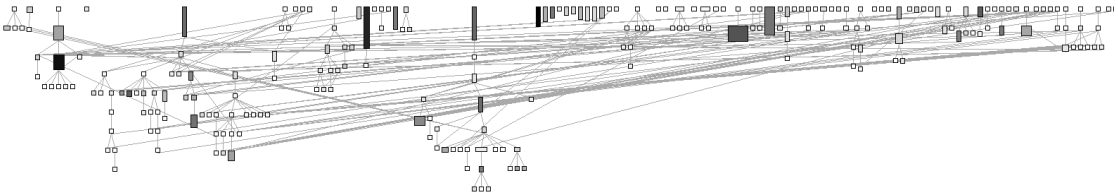

Figure 5.2: Complexity view of a multi-inheritance Lisp system

intended for single-inheritance languages, where the hierarchy is a tree, from where the name of the layout: tree layout.

Class blueprints.

We applied the class blueprint visualization on Lisp classes and we'll present here the results.

When comparing a blueprint visualization of a Java class (see Figure 2.7) with a blueprint visualization of a Lisp class (see Figure 5.3), we observe that the Lisp class blueprint is missing nodes from the first and third layer.

Figure 5.3: A blueprint visualization of a Lisp class

The first layer is the Initialization layer and contains the methods responsible with the initializing the values of the attributes of the object (the method name contains the substring *initialize* or *init* or the method is a constructor or for Smalltalk the method is placed within protocols whose name contains the substring *initialize*). Because in CLOS the initialization of the attributes is done via the *initform* option at the attribute declaration, we don't have any methods in the initialization layer.

The third layer is the Internal implementation layer and contains the methods that represent the core of a class and are not supposed to be visible to the outside world (in languages like Java and C++ if it is declared as private or the method is invoked by at least one method defined in the same class). Because CLOS is a generic function based object-oriented language, rather than a message-sending language, methods do not belong to classes, but are independent. That is why there are no class internal methods.

5.3 The Class-Method Relation View

The Class-Methods Relation view is a visual way of supporting the understanding of the relation between classes and methods in a object-oriented Lisp program. It is a 2-dimensional map representing classes, methods and the relations between them.

This view was specially designed to reveal the difference between Lisp's object-system and other object-oriented languages, like C++ or Java, and better understand the Common Lisp Object System view of object-oriented programming in contrast to other languages. The use of this view reveals unique CLOS features, not found in other object systems, like multi-methods, multiple dispatch or object dispatching.

Description. The *Class-Method Relation* view visualizes classes and methods as nodes, while the edges represent specialization relationships. The shape and color of the node encodes the node type: classes are represented by blue rectangles and methods by green circles. The metrics used to enrich this view are NOA (number of attributes) for the class width and NOM (number of corresponding methods) for the class height.

As previous mentioned in Chapter 3, in CLOS the relation between classes and methods is not one of containment, but one of specialization. Each method can specialize on one or more classes, meaning that a method can be dispatched upon each class of its parameters. As a consequence we'll have classes connected to one or more methods and methods connected with one or more classes.

The layout of the view is a *force-based layout*. The purpose of this layout is to position the nodes of the graph in two dimensional space so that all the edges are of more or less equal length and there are as few crossing edges as possible. The force-directed algorithms achieve this by assigning forces amongst the set of edges and the set of nodes; the most straightforward method is to assign forces as if the edges were springs and the nodes were electrically charged particles. The entire graph is then simulated as if it were a physical system. The forces are applied to the nodes, pulling them closer together or pushing them further apart. This is repeated iteratively for a fixed number of iteration or eventually until the system comes to an equilibrium state; i.e., their relative positions do not change anymore from one iteration to the next. At that moment, the graph is drawn. The physical interpretation of this equilibrium state is that all the forces are in mechanical equilibrium.

In Figure 5.4 we have an example for a small Lisp project with 47 classes and 275 methods. The view is obtained after a fixed number of iterations and not until the state of equilibrium, because that would need much more computing resources and would take too long.

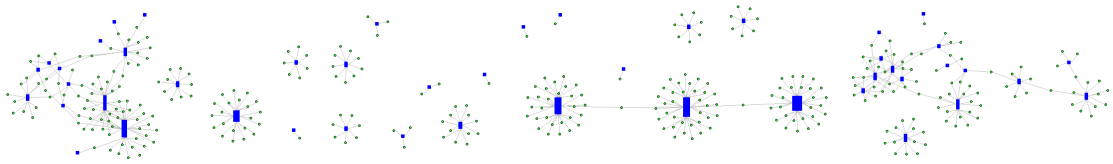

Figure 5.4: A Class-Method Relation View

Reverse engineering goals. The *Class-Method Relation* view helps to identify possible independent or loosely coupled components of the system. This way the re-engineer can study each component separately and after understanding the components can study the overall system more easily. The components can be identified on the view by looking at conglomerates of classes and methods heavily connected or at loosely coupled components that have little or even no connections to other parts of the system. The view can also help identifying degenerated classes that have no connection with any method or methods with no connection to any class.

By studying a series of view we have extracted a series of visual patterns, described bellow.

- *Stars.* Star like figures with a blue rectangle in the middle, a class, and green circles at the margin, methods, represent "classical classes", meaning that they resemble C++/Java style classes, where we have a class and a set of methods that belong to that class. In the case of CLOS, that methods are connected to only that class because they specialize on only the class of one parameter, so they are like C++/Java methods which specialize on *self*, the class they belong to. For an example of the *star* visual pattern see Figure 5.5.

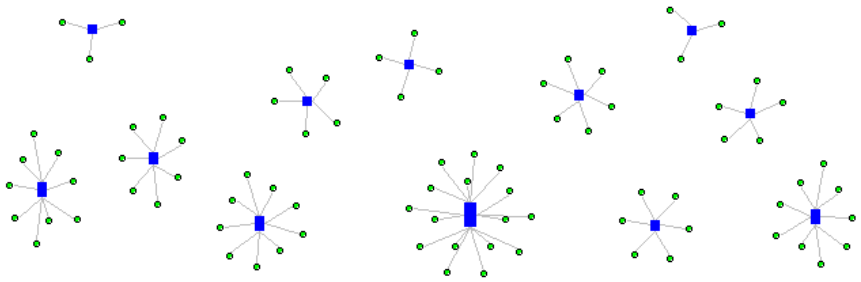

Figure 5.5: A Class-Method Relation View: classical class

- *Lonely classes.* Lonely classes are those classes that have no connection to any methods. They are identified by the blue rectangles with no lines connecting to it. This classes are degenerated classes and can be caused by one of two reasons. One possibility is that the class is only used as a data storage, similar to a C *struct*, and has no methods associated to it. Another possibility can be found in projects that make heavy use of macro, where the method definitions can be "hidden" behind a macro definition. For an example of the *lonely class* visual pattern see Figure 5.6.

- *Lonely methods.* Lonely methods are those methods that have no connection to a class. They are identified by the little green circles with no lines connecting to it. This means that the method do not specialize on any class. This possible in CLOS because here methods cans also specialize on objects. In CLOS, object identity in addition to object type can be used as a description for the method's parameters. For example, a method can be defined that operates only when the actual argument is equal (applying the Lisp function EQL) to the object in the parameter specification (these are called *eql specializations*). Neither Java, C++ nor Smalltalk support selection on anything other than the types of arguments. So the so called lonely methods are methods that specialize on objects rather than classes and that is why they are not connected to any class. For an example of the *lonely method* visual pattern see Figure 5.6.

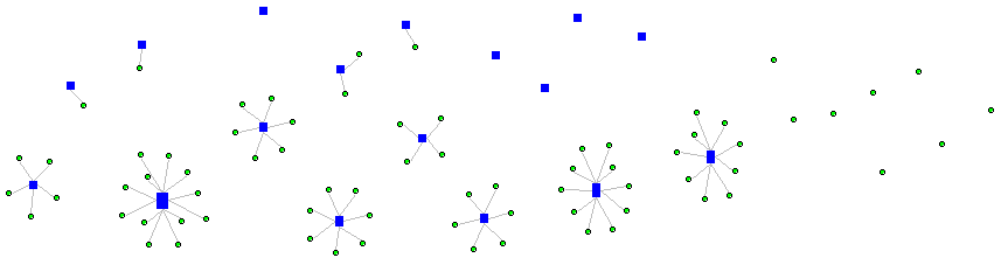

Figure 5.6: A Class-Method Relation View: lonely classes and methods

- *Conglomerates.* Conglomerates are formed by a set of classes and methods heavily connected by edges and situated in a close vicinity to each other. The classes are pulled together by methods connected to more than one class, because a method pulls together the classes that she is connected to, that she specializes on. So the result of this forces is a conglomerate of classes and methods, that gives the viewer an impression of entanglement. The methods that cause this conglomerates are called "*multi-methods*", methods that specialize on more than one class. This visual pattern is distinctive for CLOS systems, because in message-sending object-oriented languages there aren't any multi-methods. For an example of the *conglomerate* visual pattern see Figure 5.7.

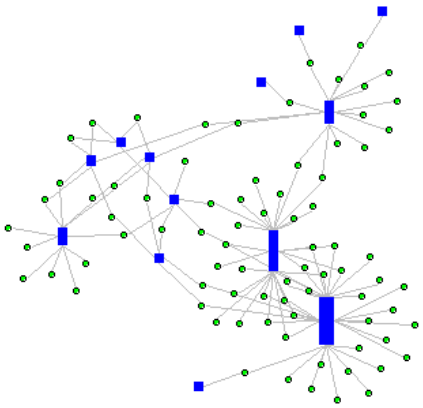

Figure 5.7: A Class-Method Relation View: class-method conglomerate

5.4 The Class Types View

The Class Types view is a visual way of identifying different types of classes, based on their structure. Classes are classified based on the attributes to methods ratio into different types, encoded with colors.

Description. The Class Types visualization has two types of views corresponding to the applied layout: a scatterplot layout and a tree layout.

Case A. The Class Types view visualizes classes as nodes placed on a scatter diagram. The position of the nodes represents the number of attributes of the class (the X axis) and the number of associated methods (the Y axis), with the origin placed on the upper left side of the diagram. The color of the nodes is based on the attributes to methods ratio, conforming with the following coloring scheme:

NOA = 0 and NOM = 0	white
NOA = 0	yellow
NOM = 0	red
NOA/NOM < 1/4	light-green
1/4 < NOA/NOM < 1	green
NOA/NOM > 1	brown

The thresholds are based on average values extracted from a series of case studies, that we'll present later in Chapter 7.

The layout of the view is a *scatterplot layout*. A scatterplot, scatter diagram or scatter graph is a chart that positions nodes in an orthogonal grid (origin in the upper left corner) according to two measurements. The data is displayed as a collection of points, each having one coordinate on the horizontal axis and one on the vertical axis. This layout algorithm is useful for comparing two metrics in large populations. Entities with two identical measurements will overlap. This layout is very scalable, because the space it consumes is due to the measurements of the nodes and not to the actual number of nodes.

In Figure 5.8 we have an example of a Class Type view with a scatterplot layout.

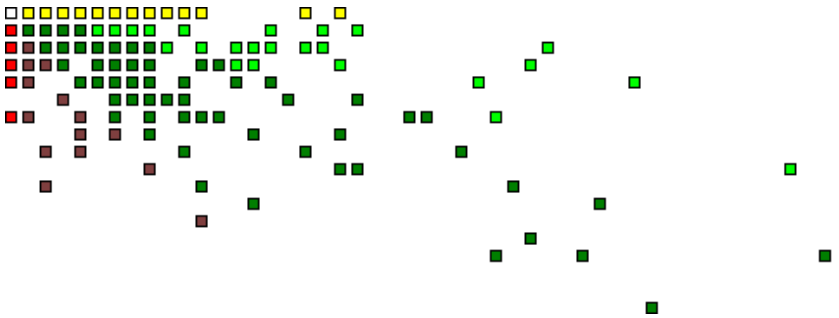

Figure 5.8: A Class Types View: scatterplot layout

Case B. We also applied to the Class Types view another layout. The second layout is an tree layout. In this case we visualize classes as nodes, while the edges represent inheritance relationships. The metrics used to enrich this view are NOA (the number of attributes of a class) for the width and NOM (the number of methods of a class) for the height. The color of the nodes is based on the attributes to methods ratio, as in the previous case.

In Figure 5.9 we have an example of a Class Type view with a tree layout.

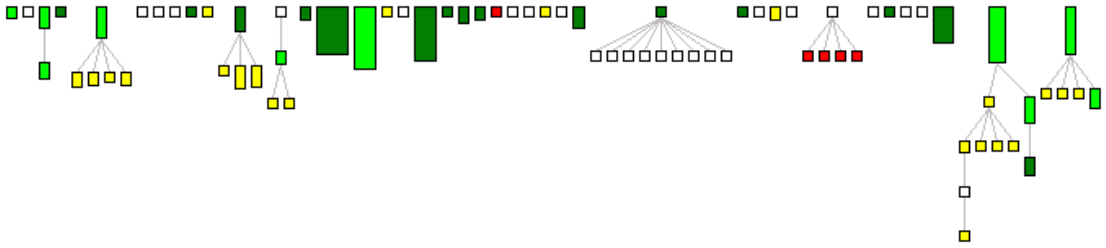

Figure 5.9: A Class Types View: tree layout

Reverse engineering goals. The *Class Types* view helps to identify and locate different type of classes, based on their structure, more precisely on the attributes to methods ratio.

Based on the coloring scheme developed we can differentiate six types of classes:

- *White classes:* are empty classes, that do not contain any attributes and don't have any associated methods. This are auxiliary classes that do not belong to the system but were introduces into the model for a better understanding of the system. This are, for example, language classes (like Object or Class) that are inherited by a application class and therefore included into the model to be able to model the inheritance relationship or classes from external libraries that were not included into the model. And because they were introduced just as auxiliary classes, for different relations to other classes, they do not have any attributes or methods.

- *Yellow classes:* are so called "mixins". A mixin is a class designed to be used additively, not independently, that it is intended to be composed with other classes or mixins. Mixins provide auxiliary behavior. The auxiliary behavior can be in the form of auxiliary methods that are combined with the primary methods of the dominant class, or it can be in the form of new primary methods. The term "mixin" (or "mixin class") was originally introduced in Flavors [Moo86], a predecessor of CLOS. The difference between a regular, stand-alone class and a mixin is that a mixin models some small functionality slice (for example, printing or displaying) and is not intended for standalone use, but it is supposed to be composed with some other class needing this functionality. One use of mixins in object-oriented languages involves classes and multiple inheritance. In this model, a mixin is represented as a class, which is then referred to as a "mixin class," and we derive a composed class from a number of mixin classes using multiple inheritance.

- *Red classes:* are classes that contain only attributes and have no associated methods. This type of classes are used only as a data storage, similar to a C *struct*.

- *Light-green classes:* are classes with a attributes to methods ratio smaller then 1/4, representing classes with few attributes and many methods. This classes have little state and a lot of associated behavior. This pattern is typical for *algorithm* classes, that have little state but a lot of behavior, doing a lot of computations.
- *Green classes:* are classes with a attributes to methods ratio between 1/4 and 1, representing classes with an average attributes to methods ratio. These are called "healthy"classes.
- *Brown classes:* are classes with a attributes to methods ratio bigger then 1, representing classes with more attributes than methods. This classes can be data storage classes that define several attributes and the methods mainly define accessors to attributes.

Besides the exact coloring scheme, we can also investigate nodes based on their positioning, in the case of the scatterplot layout. So nodes in the upper left corner represent small classes with only a few attributes and methods. They aren't usually a problem. Nodes in the opposite corner, the bottom right corner represent big classes, which contain a large number of attributes and associated methods. It may be worth looking at the internal structure of the class to learn if the class is well structured or if it could be decomposed or reorganized.

In the case of the tree layout applied to the inheritance hierarchy, we can investigate the way state and functionality is distributed along the inheritance hierarchy. For example in the Figure 5.4 we can observe that in general subclasses only provide functionality, the yellow subclasses in the left and right part of the figure, or only state, the red subclasses.

5.5 The Programming Style Distribution View

The Programming Style Distribution view is a visual way of identifying the programming paradigm used in a program. It is a visual representation of the distribution of programming paradigms over the program's packages.

Description. The Programming Style Distribution view visualizes the main entities of the program (functions, macros, global variable, classes and methods), encoded with different colors, and placed on a program package map. This layout represents a *Distribution Map.*

The *Distribution Map* [DGK06] illustrates the correlation between a chosen concept and the structural modularization of a system. It is composed of large rectangles containing small squares in different colors. For each structural module there is a large rectangle and within those for each software artifact a small square. The color of the squares refers to the concepts implemented by these artifacts. The choice of colors is crucial to the readability of the Distribution Map.

In the following lines we present a more formal definition of the Distribution Maps. Given the software system S as a set of software artifacts and two partitions P and Q of that set, the Distribution Map represents a means to visualize Q compared to P. The visualization is composed of large rectangles containing small squares in different colors. There is a small square for each element of S, the partition P is used to group the squares into large rectangles and the partition Q is used to color the squares. The partition P (the reference partition) corresponds to a well understood partition, typically the intrinsic structure of the software system (e.g., the package structure). The partition Q (comparison partition) is the result of an analysis, usually a set of clusters.

In our case the reference partition is the package structure of the program and the comparison partition is the distribution of program entities in the package structure. The entities and associated colors that are represented in this visualization are presented bellow:

Entities	Color	Paradigm
Functions	yellow	functional
Global variables	red	imperative
Classes and methods	blue	object-oriented
Macros	green	macro-style

The positioning of the packages on the view is based on a clustering algorithm (average linkage), that groups similar elements together. Using a *hierarchical clustering* we obtain a tree, called *dendrogram*, that imposes both a sort order and a grouping on its leaf elements. Traversing the dendrogram we collect its leaves and return them sorted by similarity, in that way we place similar elements near each other and dissimilar elements far apart of each other.

In Figure 5.10 we have an example of a Programming Distribution Map view.

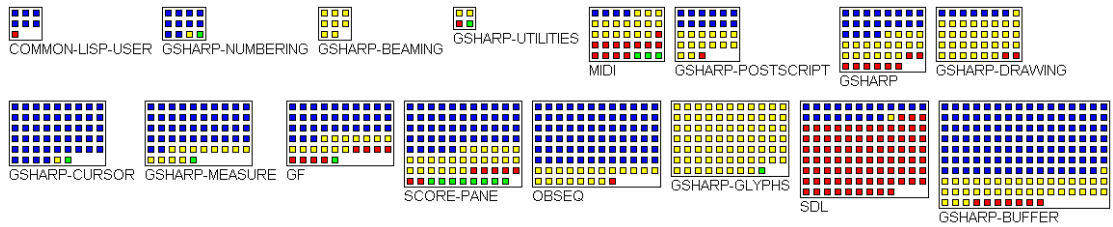

Figure 5.10: A Programming Style Distribution View: example 1

Reverse engineering goals. The *Programming Style Distribution* view helps to identify the programming style used in each package and to visualize the distribution over all the packages of the system. This way the re-engineer can study each package according to the programming paradigm used more easily. This view is also useful in determining the size and complexity of the packages and the balance between the program's packages.

By studying a series of view we have extracted two series of visual patterns: size-based, structure-based and distribution-based visual pattern.

Distribution-based visual patterns: based on the distribution of the programming styles over the program's packages. We can distinguish two patterns: distributed styles and encapsulated styles.

- *Distributed styles*: the programming paradigms are more or less evenly distributed over all the packages in the system. This is the most often case.

- *Encapsulated styles*: each style of programming is encapsulated in one or a set of packages. This is a more rare case and requires a very good delimitation of packages to be obtained.

Structure-based visual patterns: based on the structure of the packages. We can distinguish two patterns: combined styles and exclusive style.

- *Combined styles*: packages contain a mixture of styles. For example see package SCORE-PANE from Figure 5.10.
- *Exclusive style*: packages contain in majority one programming style. For example see package GSHARP-GLYPHS from Figure 5.10 which contains only yellow boxes (functions)or package SLATEC from Figure 5.11 wich contains only red boxes (global variables).

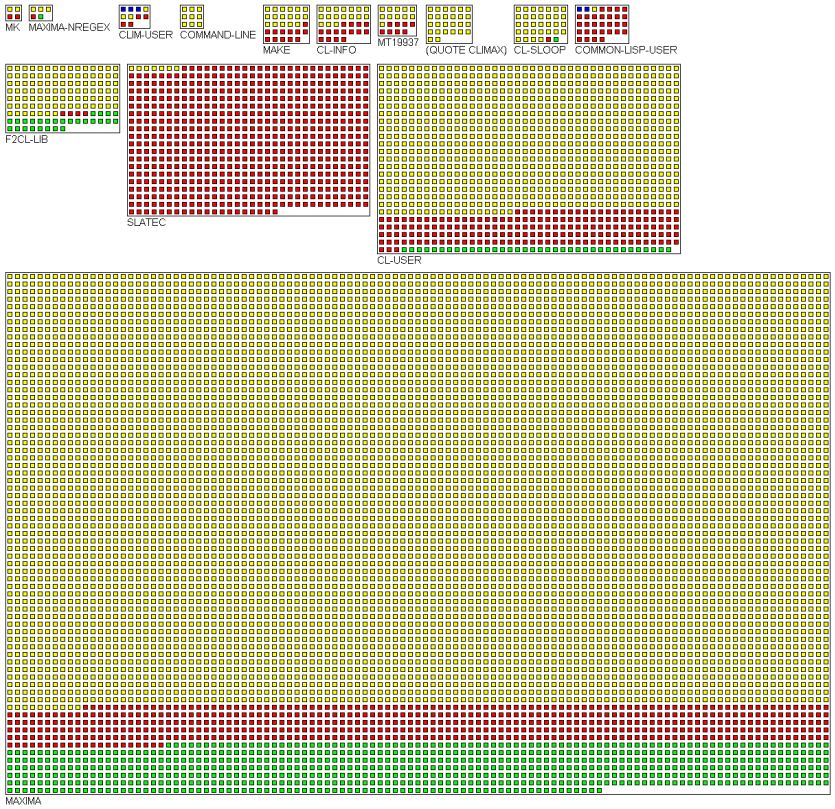

Figure 5.11: A Programming Style Distribution View: example 2

Size-based visual patterns: based on the size of the packages and the balance between packages. We can distinguish two patterns: balanced packages and monolithic package.

- *Balanced packages*: all the packages of the system are balanced in size, meaning that the program structure is well organized in packages. For examples see Figure 5.10.
- *Monolithic package*: there is one big package compared to the rest of the packages, which contains most of the system. For example see package MAXIMA from Figure 5.11. This characterizes a bad decomposition of the system in packages.

5.6 The Generic Concerns View

The Generic Concerns view is a visual way of identifying cross-cutting concerns in a system. We propose the idea that visualizing the spread of generic functions into the system can help identifying cross-cutting concerns.

Separation of concerns is a powerful principle that can be used to manage the inherent complexity of software [Kic96]. One of the benefits of separation of concerns is an increased understanding of how an application works, which helps during evolution. This benefit comes from the fact that the code belonging to a concern can be seen and reasoned about in isolation from the other concerns with which it is tangled together.

Even with proper education, understanding crosscutting concerns can be difficult without proper support for visualizing both static structure and the dynamic flow of a program. Visualizing crosscutting concerns is just beginning to be supported in IDEs, as is support for aspect code assist and refactoring. Visualization tools are important for understanding program organization and also for seeing how crosscutting concerns compose and interact. There are two approaches to present the details of crosscutting structure: tree views, and static structure diagramming approaches.

The Common Lisp Object System is based on the concept of generic functions, as seen in Chapter 3. A generic function is a collection of methods with the same name and argument structure, but with differently-typed arguments. The methods associated with the generic function define the class-specific operations of the generic function. In a generic-function system, the generic functions become the focus of abstraction; they are rarely associated unambiguously with a single class or instance; they sit above a substrate of the class graph, and the class graph provides control information for the generic functions.

The goal of Aspect-Oriented Programming (AOP) is to make it possible to deal with cross-cutting aspects of a system's behavior as separately as possible. In message-sending languages, where methods belong to classes, aspects usually cross-cut more functions from different classes and the goal of AOP is two gather the concern in one place and to allow programmers to first express each of a systems aspects in a separate and natural form and then automatically combine those separate descriptions into the final code. But in generic-function systems this is more easily achieved because the aspect can be abstracted out into a generic function, which will have more implementing methods for each class that presents that aspect. This is why we think that identifying cross-cutting, in CLOS systems, is closely related with the analyze of generic functions, calling them *generic concerns*. And consequently we propose the idee that visualizing generic functions spread over a system can help identifying cross-cutting concerns.

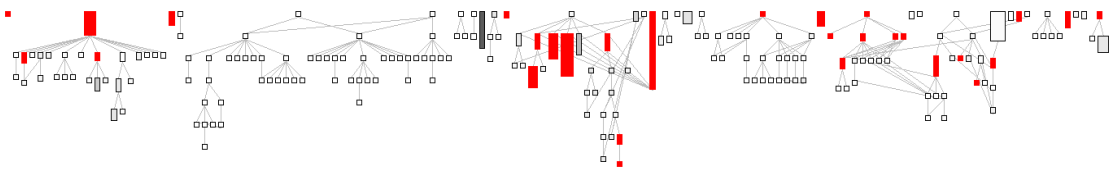

Figure 5.12: A Generic Concerns View: tree layout

Description. The GENERIC CONCERNS visualization has two types of views corresponding to the applied layout: a tree layout and a distribution map.

Case A. The GENERIC CONCERNS view visualizes the system's class inheritance hierarchy with a tree layout; classes are nodes and the edges represent inheritance relationships. The metrics used to enrich this view are NOA (the number of attributes of a class) for the width and NOM (the number of methods of a class) for the height. For each generic function we have an distinct view. The color of the nodes represents if there is a method associated with the generic function that has a class specific implementation. So in red we have the classes that are dispatched upon by one of the methods of the generic function.

In Figure 5.12 we have an example of a Generic Concern view with a tree layout.

Case B. We also applied the GENERIC CONCERNS visualization on a distribution map. This view visualizes the systems packages as large boxes and inside them we have nodes representing the package's classes. For each generic function we have an distinct view. The color of the class nodes represents if there is a method associated with the generic function that has a class specific implementation. So in red we have the classes that are dispatched upon by one of the methods of the generic function.

In Figure 5.13 we have an example of a Generic Concern view applied on a distribution map.

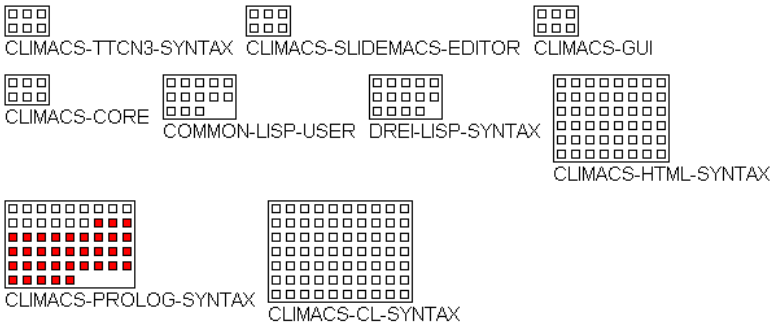

Figure 5.13: A GENERIC CONCERNS View: package distribution map

Reverse engineering goals. The *Generic Concerns* view helps to identify and locate cross-cutting concerns associated with generic functions.

By studying a series of view we have identified three types of concerns, based on the visual patterns they produce.

- *Scattered concerns*: concerns that are scattered all over the inheritance hierarchy and all over the system packages. They are general concerns that are common for most of the classes, like instantiation, drawing (in case of a graphical application), parsing (in an parser), etc. For an example see Figure 5.12 for a tree layout or Figure 5.14 for a package distribution map.

- *Balanced concerns*: concerns that are equally distributed over the system packages. They represent a common concern for all the package that includes them. For an example see Figure 5.14.

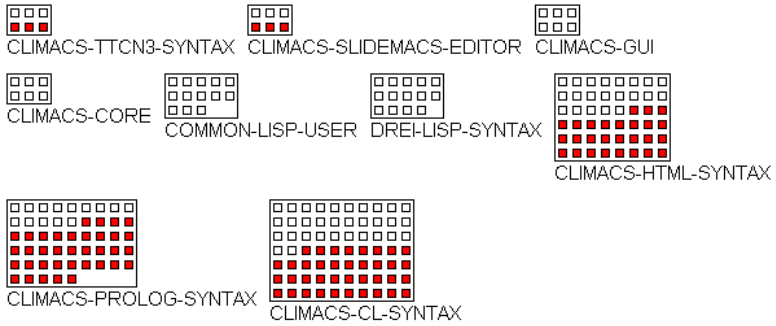

Figure 5.14: A Generic Concerns View: package distribution map

- *Localized concerns*: concerns that are only spread in one subtree of the inheritance hierarchy, see Figure 5.15, or only in one package, see Figure 5.13. They are closely related with the root of the subtree they appear in and are specific to the package they appear in.

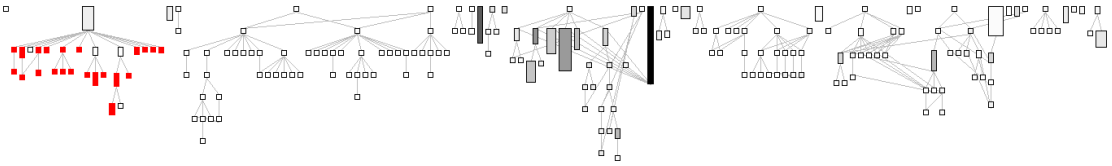

Figure 5.15: A Generic Concerns View: tree layout

Chapter 6

Tool Support

"Give us the tools, and we will finish the job."
Winston Churchill

In this chapter we'll present our tools that implement the FAMIX-Lisp meta-model and the Lisp visualizations presented in the previous chapters. First we have a model extractor (*ModeLisp*) that extract FAMIX-Lisp models from Lisp systems and then we have a Lisp plugin for the MOOSE environment (*MoosLi*), that can import FAMIX-Lisp models and browse the model and apply different analysis and visualizations.

6.1 ModeLisp: Lisp Model Extractor

ModeLisp is a model extractor, that extract FAMIX-Lisp complaint models from Lisp systems. ModeLisp is implemented in Lisp, combining functional programming, for scanning and identifying entities in source code, and object-oriented programming, for the internal data representation of the extracted entities and the associated operations for each type of entity.

ModeLisp receives as entry the directory of the project and produces a FAMIX-Lisp complaint model of the system, which can be exported in the MSE format.

The creation of the model is done in a series of stages.

- scanning the source code, reading the program entities and creating instances of their class;
- creating the relations between the entities, from the previous stage;
- resolving all the references between entities and relations (some attributes are references to other objects)
- creating the macro-expansions;
- calculating size/complexity metrics for the behavioral entities.

ModeLisp is not a toy implementation; it was successfully applied for extracting models from an important number of "real-world" systems ranging in size from 5KLOC to large-scale systems like SBCL(365KLOC) or CL-HTTP(309KLOC).

6.2 MoosLi: a Lisp plugin for Moose

MoosLi is a Lisp plugin for the Moose environment. MOOSE [NDG05] is an reengineering environment designed to provide the necessary infrastructure for building new tools and for integrating them. Moose centers on a language independent meta-model, and offers services like metrics evaluation, grouping, querying, navigation, and meta-descriptions. Several tools have been built on top of Moose dealing with different aspects of reengineering like: visualization, evolution analysis, semantic analysis, concept analysis or dynamic analysis.

Moose uses a layered architecture: information is transformed from source code into source code models, conforming with an internal defined meta-model, and stored in a repository; the environment provides some basic analysis tools and on top of them there are numerous tools developed. In Figure 6.1 we have the architecture of Moose, including the MoosLi plugin.

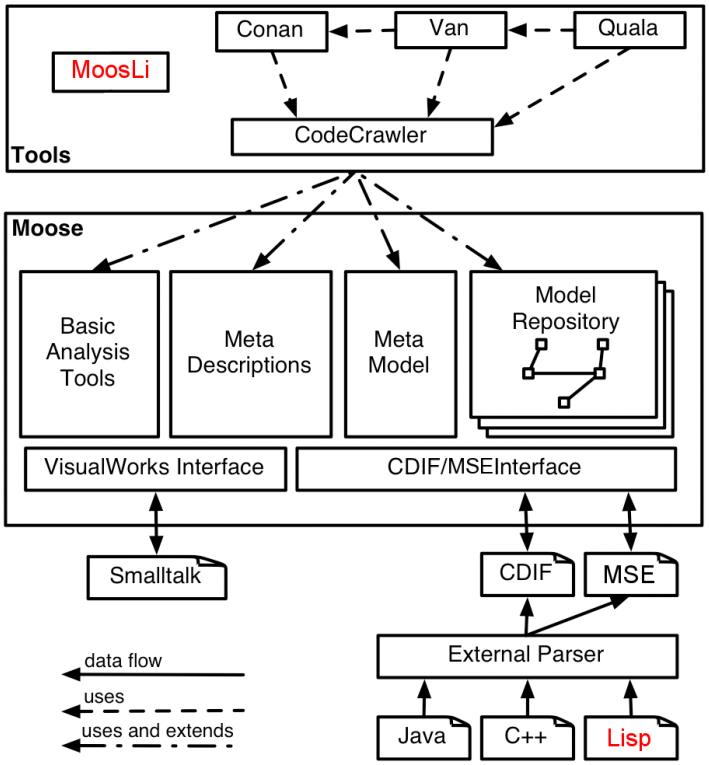

Figure 6.1: The Moose architecture, including the MoosLi plugin

MoosLi is a Smalltalk package that, when loaded into the MOOSE image, offers the capability of importing FAMIX-Lisp models in MSE format, browse Lisp models and apply different analysis and visualizations on Lisp models.

The meta-model. The FAMIX meta-model behind Moose is implemented as a series of meta-annotations into the source code, that are read at initialization when the meta-model is build. Consequently the MoosLi plugin extends the FAMIX meta-model to the FAMIX-Lisp meta-model by adding the corresponding meta-annotations for the new entities and the new attributes of the existing entities. This extension does not affect the environments initial capabilities of modeling different languages.

In addition the MoosLi plugin also offers browsing capabilities for Lisp models. This is also implemented as a set of annotations, for each group of entities and list of proprieties of entities and available operations on each type of entity.

The visualizations. As we have seen in Chapter 5, MoosLi also offers a set of visualizations, specially tailored for Lisp systems. We only enumerate them here:

- the Class-Method Relation View (section 5.3)
- the Class Types View (section 5.4)
- the Programming Style Distribution View (section 5.5)
- the Generic Concerns View (section 5.6)

All our visualizations were developed using Mondrian [MGL06], an information visualization engine, implemented as a Moose tool, that supports on-the-fly prototyping of visualizations. Our visualizations are interactive, the user can not only see but also interact (zoom, select, move, inspect, etc.) with the views.

The environment. In Figure 6.2 we have the Moose environment with the MoosLi plugin installed. In the left pane we have the models of different analyzed software systems. If we select a particular model we'll get in the second pane the groups of entities belonging to the selected model (functions, macros, classes, methods, generic functions, etc). By selecting one of the groups we'll obtain a new pane with the list of entities from the selected group. The new pane has at the bottom a filter field in which we can enter a condition which we'll be applied to the list of entities, filtering only the ones that fulfill that condition. By selecting one entity from the group, we'll obtain yet another pane with the attributes of that entity and also it's properties. This process of browsing and viewing new groups, entities or properties by adding new panes to the right can go on infinitely.

Each entity or group from the system has an associated contextual menu, obtained by right clicking on that entity or group. This contextual menu will present all the possible actions that can be applied to the selected entity. In this contextual menu we have a MoosLi group that contains a set of views that can be applied to Lisp projects. From this sub-menu we can apply the visualization presented in Chapter 5. The browser is general enough to work with any type of entity, and therefore, whenever a tool extends the meta-model with a particular entity, it is provided with the default browsing capabilities.

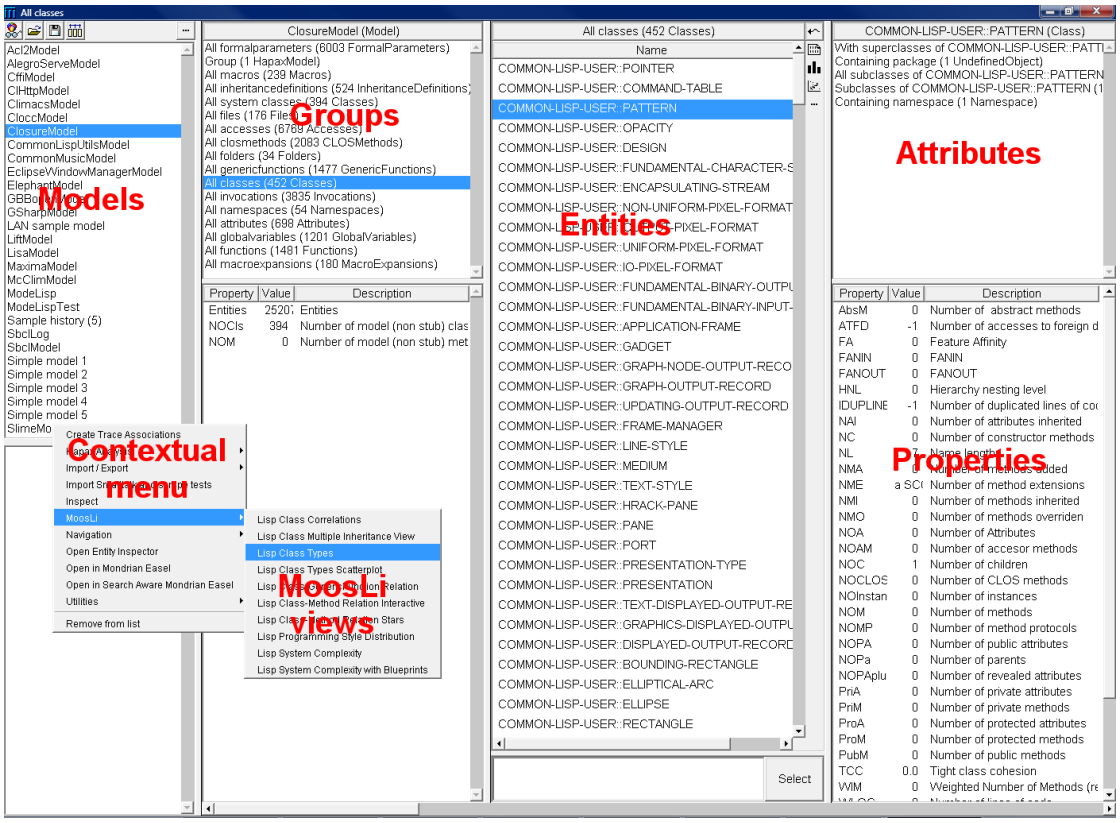

Figure 6.2: The MOOSE environment with MoosLi plugin

6.3 Visual browsers

A visual browser is a visual support for browsing software systems based on their class structure. Our visual browser has two panes. In one of them we visualize the system based on the class inheritance hierarchy with a tree layout and enriched with metrics (NOA for class width and NOM for height). In the second pane we have a list of behavioral entities (generic functions or methods) in association with the selected class from the first pane. The scope of the visual browser is to help in the exploration of the systems classes, generic functions and methods and the relation between them. The browser is a interactive browser; any entity in can be selected and inspected in detail. Also by selecting an entity from any pane, automatically the other pane will be updated to show the information in correspondence with the selected entity. We have two version of the visual browser, presented bellow: the Class-Generic Function Browser and the Class-Method Browser.

The Class-Generic Function Browser has two panes, in the left pane we have the system's inheritance hierarchy and in the right pane the list of generic functions. By selecting a generic function from the right pane, automatically in the left pane the classes that are dispatched by one of the classes corresponding to the selected generic functions are colored in red. By selecting any class from the class hierarchy in the left pane automatically in the right pane we get the list of the generic functions that have an associated method that dispatches on the selected class.

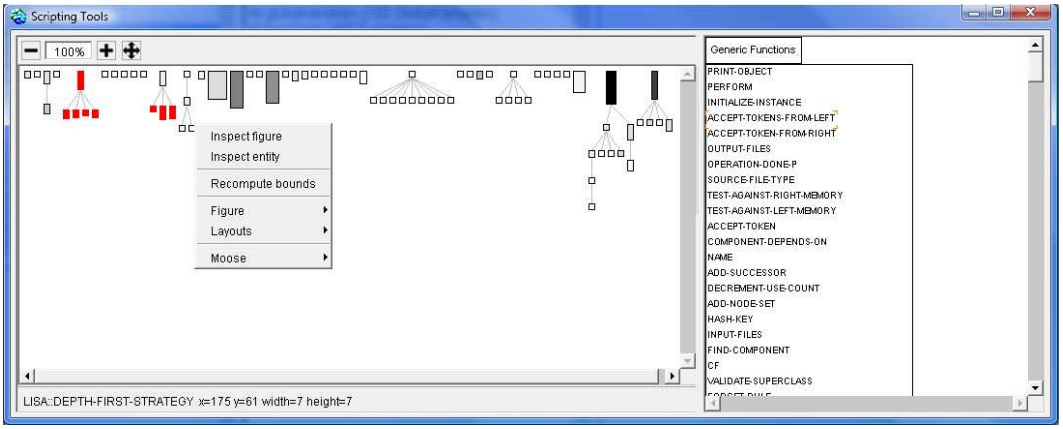

Figure 6.3: A Class-Generic Functions visual browser

The Class-Method Browser has two panes, in the left pane we have the system's inheritance hierarchy and in the right pane the list of system's methods. By selecting a method from the right pane, automatically in the left pane the classes that are dispatched by the selected method are colored in red. By selecting any class from the class hierarchy in the left pane automatically in the right pane we get the list of the methods that dispatch on the selected class.

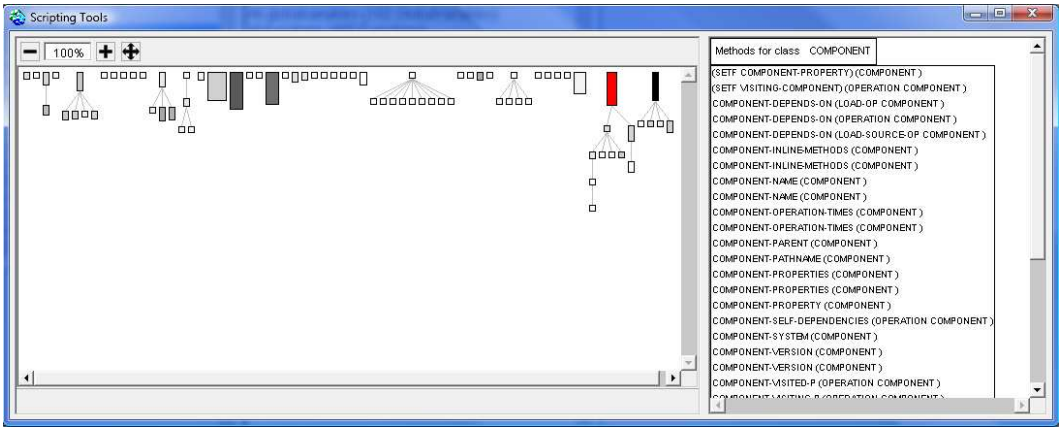

Figure 6.4: A Class-Method visual browser

Chapter 7

Case Studies

"It always seems impossible until its done."
Nelson Mandela

7.1 Overview

This chapter presents the results we found by applying our approach to several case studies. We have chosen several open-source active Lisp projects ranging in size from 5 KLOC to large projects of 365 KLOC. The projects are chosen from a variety of domain from graphical libraries, artificial intelligence projects, musical applications, web servers, mathematical software, windowing systems, interface managers, editors to compilers. The projects were chosen so that they are of various sizes, from small projects to the largest open-source Lisp projects, and represent all the programming paradigms used in Lisp: functional, imperative, macro and object-oriented programming. We have projects in which functional style is predominant, also containing a large number of macros (older Lisp mathematical software), and also object-oriented projects (interface managers or web servers).

In table 7.2 we enumerate our case studies, that we have done to validate our work, with a series of additional information extracted from the models of the analyzed systems. As a result of our studies we have also extracted a series of statistics, presented in Table 7.1.

Table 7.1: Case studies statistics

Number of extracted entities per 10 line of code	3
Number of macros per 1000 line of code	3
Number of classes per 1000 line of code	3.5
Number of functions per 1000 line of code	20
Number of methods per class	6
Number of attributes per class	2
Percent of multi-dispatching methods	8.8%
Number of methods per attribute	3

Table 7.2: Case studies

Project	Lines	Entities	Macros	Classes	Methods	Functions
SBCL	365314	42787	693	95	571	4449
CL-HTTP	309006	68041	617	883	5068	4243
CLOCC	221216	66925	728	202	993	4260
ACL2	219957	44537	542	0	0	3688
Maxima	211630	74667	777	0	0	7226
McClim	138891	53721	354	804	4408	2095
Closure	79500	25207	239	394	2083	1481
CommonMusic	59020	10113	101	14	176	1117
Eclipse Window Manager	40855	12669	150	23	166	921
GBBOpen	25158	4731	62	3	14	473
Slime	21453	3256	44	17	36	521
AlegroServe	16508	3566	55	48	275	288
Elephant	14329	6358	29	60	449	250
Lisa	13443	5174	54	77	354	460
Climacs	8370	3968	24	206	409	101
GSharp	8245	4603	17	50	379	269
ModeLisp	5177	1525	7	29	230	78

From the above list of case studies we have selected two projects for a detailed analyze, that we'll present in this chapter. The case studies we selected are a Lisp compiler (SBCL) and an artificial intelligence software agent (Lisa). We have chosen this two examples because they represent both medium and large systems, they are active projects and they combine functional, imperative, macro and object-oriented programming. To test our approach we apply our newly developed visualization, presented in Chapter 5, on the models extracted from the chosen projects and analyze the results to draw the conclusions.

7.2 Case study 1: SBCL

Steel Bank Common Lisp (SBCL) [sbc] is an open-source (free software) compiler and runtime system for ANSI Common Lisp, derived from the CMUCL system (Carnegie Mellon University Common Lisp). It provides an interactive environment including an integrated native compiler, a debugger, and many extensions. SBCL runs on a number of platforms, like Linux, MacOSX, Solaris, FreeBSD or Win32 and on a variety of hardware architectures, like x86, 64-bit x86, PowerPC, SPARC, Alpha or MIPS.

SBCL is a fork off of the main branch of CMUCL. SBCL is distinguished from CMUCL by a greater emphasis on maintainability. In particular, the SBCL system can be built directly from its source code, so that the output corresponds to the source code in a controlled, verifiable way, and arbitrary changes can be made to the system without causing bootstrapping problems. SBCL also places less emphasis than CMUCL does on new non-ANSI extensions, or on backward compatibility with old non-ANSI features.

In Table 7.3 we present the numbers extracted from the SBCL project.

Table 7.3: SBCL project statistics

Project size (in MB)	15.8
Lisp source code size (in MB)	12.6
Extracted model size (in MB)	8.2
Number of lines	365314
Number of source code files	680
Number of extracted entities	42787
Number of packages	60
Number of functions	4449
Number of global variables and constants	1091
Number of macros	693
Number of classes	95
Number of generic functions	472
Number of methods	571
Number of extracted entities per 10 line of code	1.2
Number of macros per 1000 line of code	2
Number of classes per 1000 line of code	3
Number of functions per 1000 line of code	16
Number of methods per class	6
Number of attributes per class	1
Number of methods per generic function	1
Number of methods per attribute	5

SBCL is one of the largest open-source Lisp projects. In this project functional programming (almost five thousand functions) and imperative programming (a thousand global variables and constants) are the main paradigms used, while object-oriented programming is only marginally used (under one hundred classes). This is somehow understandable, because usually compiler don't have and object-oriented structure. When comparing the relative numbers from the second part of the Table 7.3 with the average values for Lisp projects from Table 7.1 we observe that the number of different entities per line of code is smaller than average. We observed this situation on all the large projects, were there are less entities per line of code, but they are larger in size and also contain more comments, because of the bigger complexity of the code. On the object-oriented size we observe that classes have on average only one attribute and six methods. So classes have less state and more functionality.

Now we'll take each visualization from Chapter 5 and apply it to the current project and explain the result.

In Figure 7.1 we have the Programming Style Distribution view of the SBCL project.

Looking at Figure 7.1 there a series of observations that can be made. First, we observe that we have two types of packages: packages that have one predominant programming style (*exclusive style* visual pattern), like SB-GRAY or SB!BIGNUM and to a lesser extend ASDF, and packages

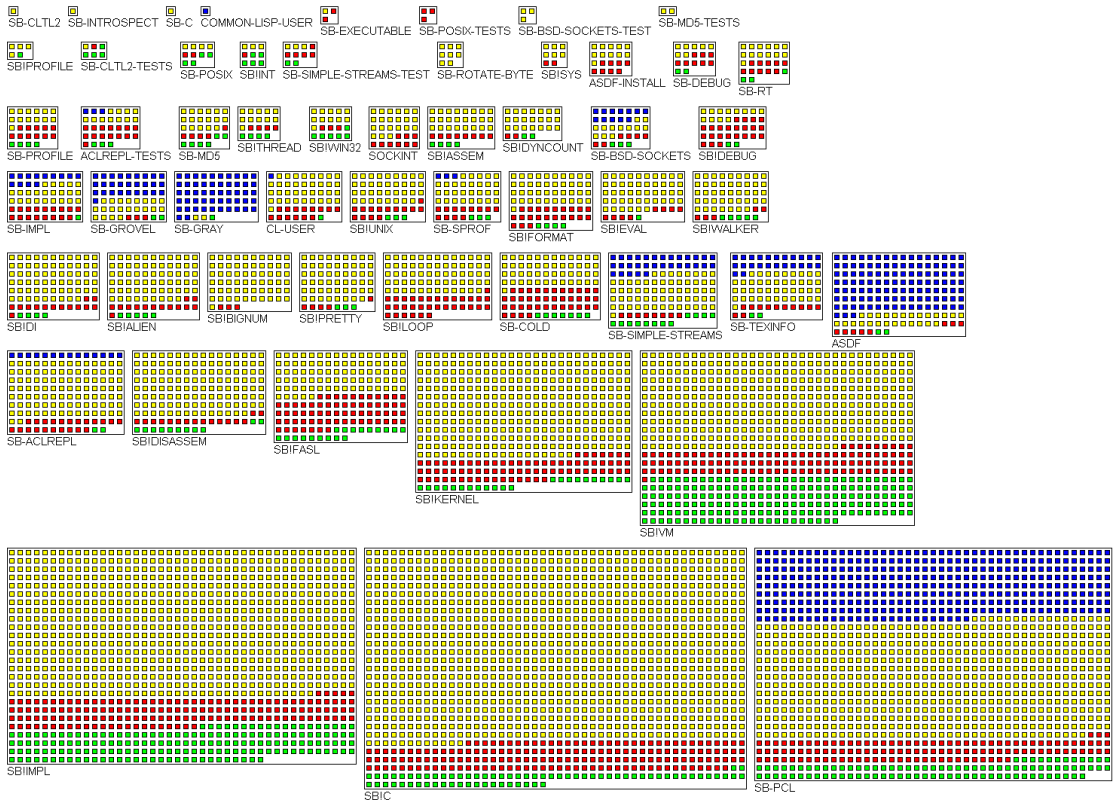

Figure 7.1: SBCL: Programming Style Distribution View

that combine more programming styles (*combined styles* visual pattern), like SB-PCL, SB!C, SB!IMPL or SB!VM.

When we take a look at the object-oriented code distribution, represented by the blue boxes, we observe that it is contained to couple of packages (*encapsulated style* visual pattern). The most of the object-oriented code can be found in the following packages: SB-PCL, ASDF and SB-GRAY. SB-PCL is the package responsible with the definition of Portable Common Lisp. PCL (Portable Common Loops) is a portable CLOS implementation. So its obvious why this package contains more object-oriented code. ASDF (Another System Definition Facility) is a system definition utility; it fills a similar role for Common Lisp as *make* does for C. It intends to solve the same basic class of problems as *mk-defsystem*, an older Lisp system definition utility, but internally it takes advantage of modern Common Lisp features and it uses CLOS for extensibility. So again is obvious why this package is in majority object-oriented. The third object-oriented package, SB-GRAY, implements gray streams. Gray stream, named after David N. Gray, are a set of extensible object-oriented version of Lisp streams. They were introduced from the inability to customize or extend stream behavior. So once again is clear why this package is almost exclusively object-oriented.

If we study the distribution of macros over the packages, represented by the green boxes, we notice that most of them are in two packages: SB!VM and SB!IMPL. SB!VM is the package responsible with the definition of the Lisp virtual machine, that provides a common interface for all the computer architectures supported by SBCL. This is a perfect example of macro use: implementing a domain specific language, in our case the VM. The second package that contains a lot of macros, SB!IMPL is responsible with the implementation of the language, the language definition. Being a metacircular language, the Lisp language definition is implemented in term of Lisp macros. As a curiosity even Lisp macros are implemented in term of macros. Most of the language's interface is implemented in term of macros. This was one of the reasons we have chosen to analyze this system, a Lisp compiler written in Lisp.

In Figure 7.2 we have the Class-Method Relation view of the SBCL project.

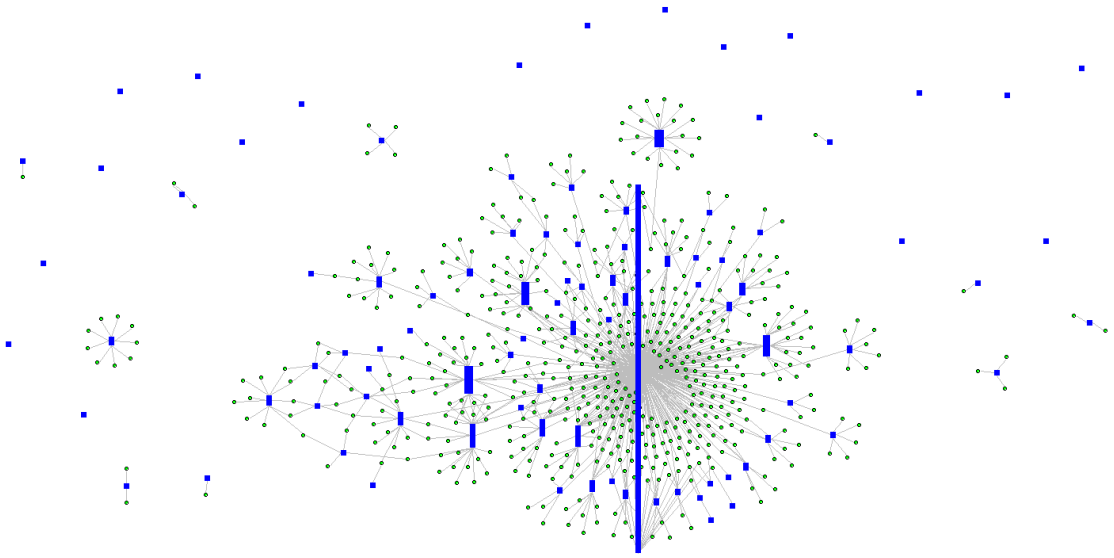

Figure 7.2: SBCL: Class-Method Relation View

When we applied the Class-Method Relation view on the SBCL project we obtained the most interesting result, seen in Figure 7.2. We see a giant class in the center of the view and the rest of the classes were drawn to the giant class. It is visually similar with a galaxy, with the giant class being the center of the galaxy and the other entities orbiting around it. This is the perfect example of the *conglomerate* visual pattern. Off-course the most interesting class of the system is the giant class. As we have seen most of the object-oriented code is in the PCL package, responsible with the implementation of Lisp's object system. When examining that package we find that the giant class is the T class. T is the root of all types and classes in Lisp and also CLOS, similar with the Object class from Java. The classes that form the conglomerate from the center of the view belong to the SB-PCL package. Besides the giant T classes, other major classes are OBJECT, CLASS, SLOT, GENERIC-FUNCTION or METHOD, classes representing the main entities from CLOS. The small scattered classes in the margin of the view are the classes belonging to other packages than SB-PCL.

In Figure 7.3 we have the Class types view, with scatterplot layout, of the SBCL project.

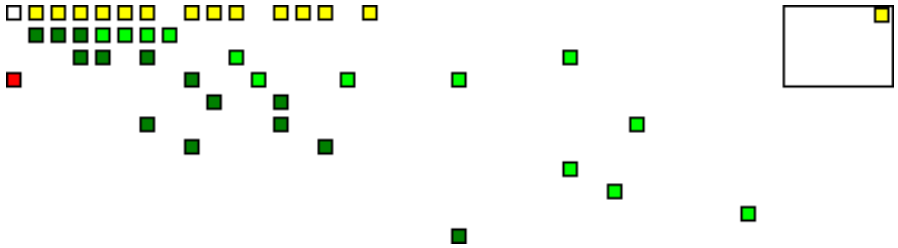

Figure 7.3: SBCL: Class Types View - scatterplot layout

When studying the distribution of class types from the SBCL project in Figure 7.3 we observe that most of the classes have more functionality than state, being yellow or light-green. In the top right corner there is a snippet representing the T class, which is really very far on the X axis, the functionality axis, meaning that the T class has no state but a large number of methods associated with it. This normal, because T is the root of the system hierarchy and most of the generic function should have an implementation method that is specialized on the root. There is only one red box, that only has state and no functionality, that is a test class containing some teste cases represented as attributes.

In Figure 7.4 we have the Class Types view, with tree layout, of the SBCL project.

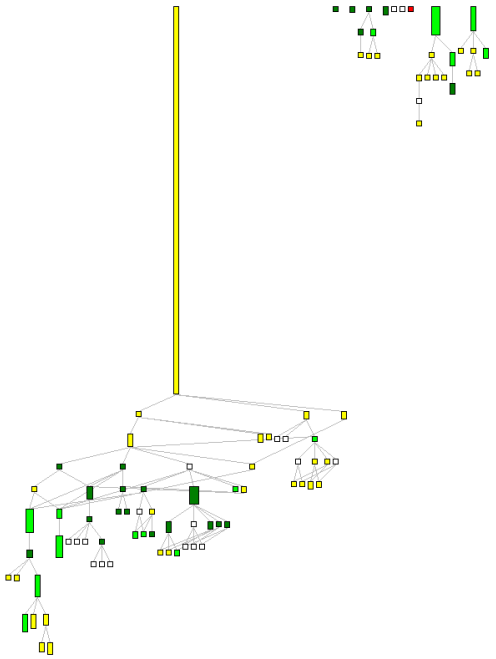

Figure 7.4: SBCL: Class Types View - tree layout

When we take a look at the Class Types view based on the tree layout, the giant T class is again in the center of attention, the very tall yellow bar. Now we see that T is the root of the class hierarchy representing the implementation of the object system. The classes in the upper right belong to the other packages and that are not connected with the implementation of the object system. Another thing we observe is the fact that the upper classes from the main tree are yellow, meaning that they have only functionality and no state at all. They are similar with *abstract* classes from Java or C++, with the difference that in our case the methods belonging to these classes do have implementations, unlike the abstract classes from C++ or Java, which only define an interface to a class. When analyzed in more detail the main inheritance tree is very similar with the CLOS Metaobject Protocol hierarchy [KdRB91], that every CLOS implementation contains.

In Figure 7.5 we have a Generic Concerns view of the SBCL project, with two layouts: a tree layout in the left and a package distribution map on the right.

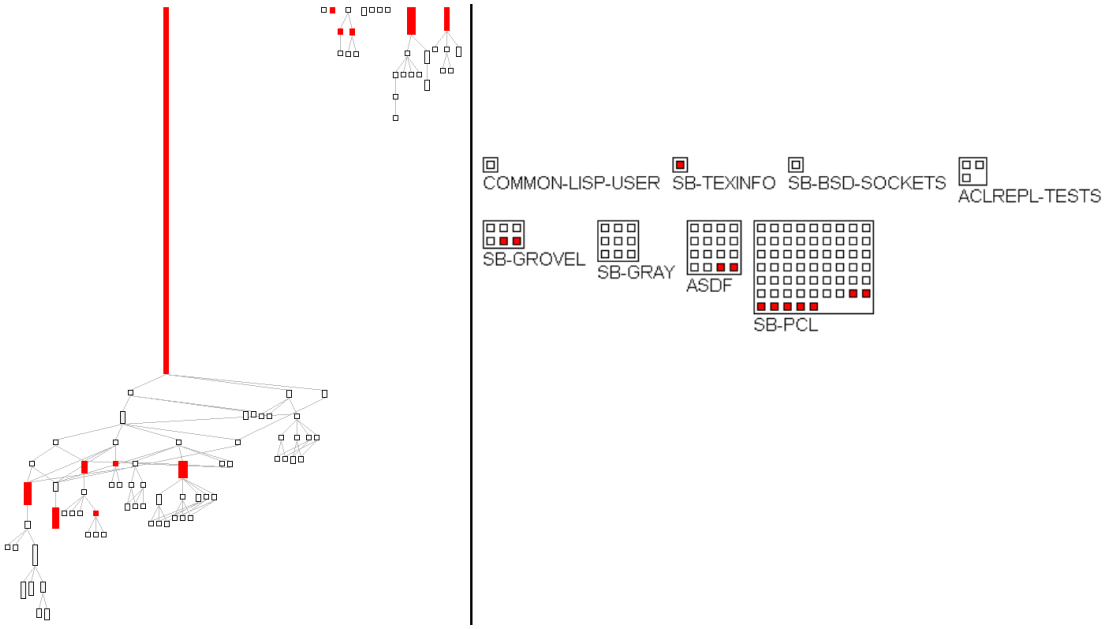

Figure 7.5: SBCL: Generic Concerns Views: scattered concern

In Figure 7.5 we can identify a *scattered concern*, that is present in all parts of the system's class hierarchy and distributed in more of the system's packages. The concern is present in all the sub-trees of the hierarchy and in most of the packages, so we can say that this concerns is a general one. The visualized generic concern is the *print-object* generic function. The generic function *print-object* writes the printed representation of an object to a stream. The function print-object is called by the Lisp printer and it should not be called by the user. This function has a default implementation, applicable for all types of objects, and for each class the programmer can provide a specific implementation based on the class specifics. So it is normal to see this concern spread all over the system.

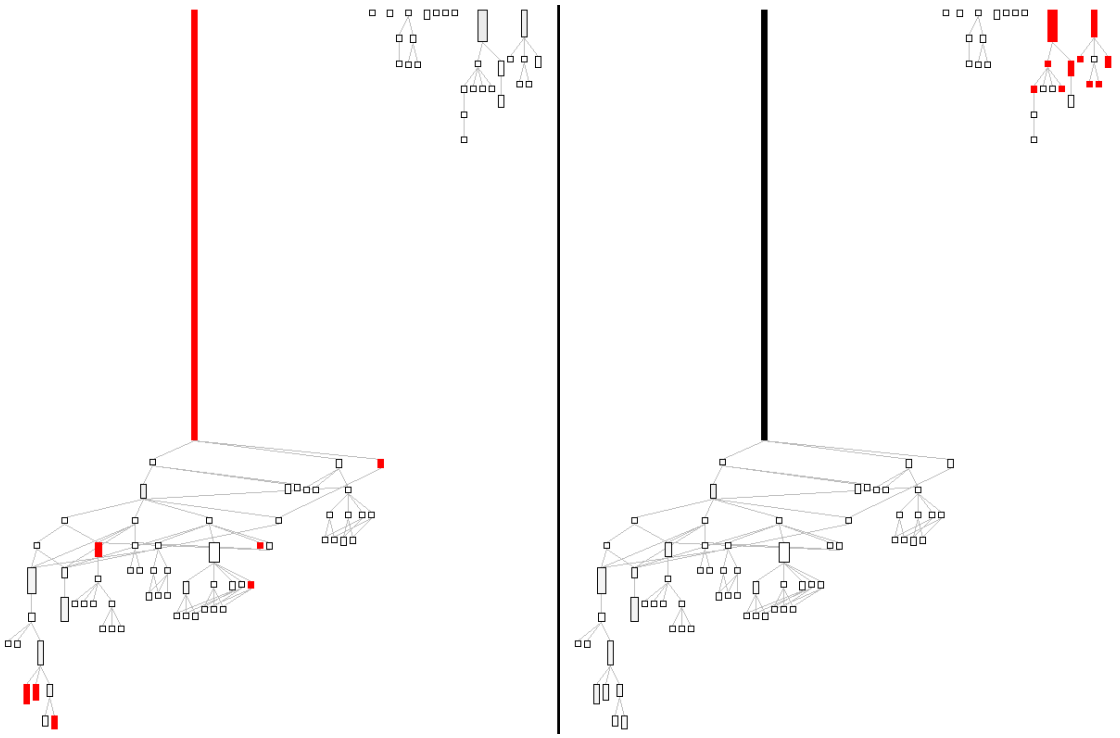

Figure 7.6: SBCL: Generic Concerns Views: localized concerns

In Figure 7.6 we can see two *localized concerns*, that only spreads on one sub-tree of the system's class hierarchy. On the left we have a generic concern that spread in the main sub-tree, that represents the implementation of the object system. The viewed generic concern is *documentation*. Each Lisp entity definition has an option called *:documentation* that causes a documentation string to be attached to the entity's name. The generic function documentation returns the documentation string associated with the given object if it is available. This is a system defined function and user should not define any extensions. That is the reason why this concern appears only in the main sub-tree, the one that is implementing Lisp's object system and implicitly the documentation option and generic function, and the other sub-trees don't have this concern.

In Figure 7.6, on the right side we a generic concern that only spread in the two upper right sub-trees. Those two sub-trees belong to the ASDF package, that implements a system definition utility. The visualized generic concern is the *perform* generic function. The ASDF system definition utility talks in terms of components (part of the system) and operations on components (compile, load, etc). Each operation can be performed on a node, work done by the perform generic function. This is the reason perform is only spread in the ASDF package and is present on most of the classes in that package.

7.3 Case study 2: Lisa

The Lisa Project [lis] is a platform for the development of Lisp-based Intelligent Software Agents. Lisa is a production-rule system implemented in the Common Lisp Object System (CLOS), and is heavily influenced by CLIPS and the Java Expert System Shell (JESS). At its core is a reasoning engine based on an object-oriented implementation of the Rete algorithm, a very efficient mechanism for solving the difficult many-to-many matching problem. Intrinsic to Lisa is the ability to reason over CLOS objects without imposing special class hierarchy requirements; thus it should be possible to easily augment existing CLOS applications with reasoning capabilities. As Lisa is an extension to Common Lisp, the full power of the Lisp environment is always available. Lisa-enabled applications should run on any ANSI-compliant Common Lisp platform.

In Table 7.4 we present the numbers extracted from the Lisa project.

Table 7.4: Lisa project statistics

Project size (in MB)	2.7
Lisp source code size (in MB)	0.5
Extracted model size (in MB)	1
Number of source code files	68
Number of lines	13443
Number of extracted entities	5174
Number of packages	13
Number of functions	460
Number of global variables and constants	102
Number of macros	54
Number of classes	77
Number of generic functions	240
Number of methods	354
Number of extracted entities per 10 line of code	3.8
Number of macros per 1000 line of code	4
Number of classes per 1000 line of code	5.7
Number of functions per 1000 line of code	34
Number of methods per class	5
Number of attributes per class	2
Number of methods per generic function	1
Number of methods per attribute	3

Lisa is a medium to small sized project. In this project object-oriented programming is almost the predominant paradigm (with 77 classes and almost as many methods as functions). This is explained by the fact that at its core Lisa is a reasoning engine based on an object-oriented implementation of the Rete algorithm. When comparing the relative numbers from the second part of the Table 7.4 with the average values for Lisp projects from Table 7.1 we observe that the number of different entities per line of code is higher than average. We observed this situation

on all the small projects, were there are more entities per line of code, but they are smaller in size and contain less comments. On the object-oriented size we observe that the projects fits in the average numbers extracted from the other case studies.

Now we'll take each visualization from Chapter 5 and apply it to the current project and explain the result.

In Figure 7.7 we have the Programming Style Distribution view of the Lisa project.

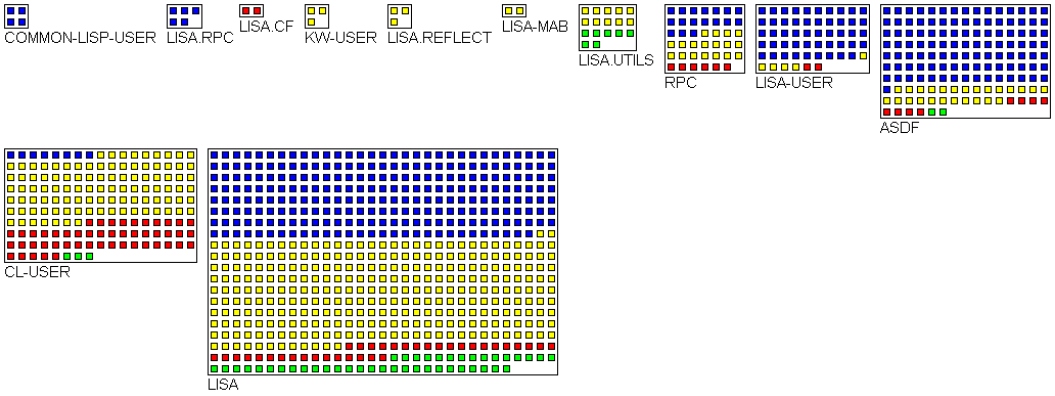

Figure 7.7: Lisa: Programming Style Distribution View

Looking at Figure 7.7 there a series of observations that can be made. First, we observe that we have two types of packages: packages that have one predominant programming style (*exclusive style* visual pattern), like LISA-USER or the small packages and to a lesser extend ASDF, and packages that combine more programming styles (*combined styles* visual pattern), like LISA or CL-USER.

When analyzing the distribution of the system's entities in packages we can see that with the exception of a few smaller packages that are external libraries used in the project, most of the system is found in one big package, the LISA package. This can be an example of the monolithic package visual pattern if we would disregard the utilities packages.

When we take a look at the object-oriented code distribution, represented by the blue boxes, we observe that it is distributed in most of the system's packages (*distributed style* visual pattern). The most of the object-oriented code can be found in the following packages: LISA, LISA-USER and ASDF. As we have seen in the SBCL case study, ASDF (Another System Definition Facility) is object-oriented extensible system definition utility. In the LISA package, a large package containing more than half of all the systems entities, we have the object-oriented implementation of production-rule system and the LISA-USER package contains all of Lisa's exported symbols, making up the user programming interface of the system. The little packages, with few entities, are small isolated features of the system, like LISA.RPC (remote object reasoning through remote procedure calls), LISA.REFLECT (wrapper functions that provide MOP functionality), LISA.CF (the uncertainty mechanism).

If we study the number of macros, represented by the green boxes, and their distribution over the packages we notice that this project is not making a big use of them and that most of the macros are in the LISA package (*encapsulated style* visual pattern).

In Figure 7.8 we have the Class-Method Relation view of the Lisa project.

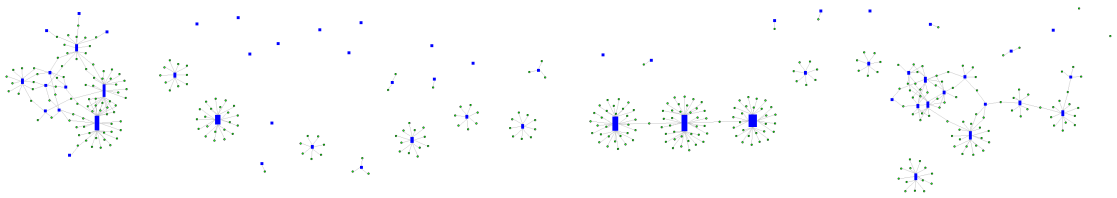

Figure 7.8: Lisa: Class-Method Relation View

When we take a look at the Class-Method Relation view from Figure 7.8 we observe tree interesting things: the two conglomerates from the left and right of the view and the tree big classes in a chain in the center of the view. The *conglomerate* pattern from the left of the figure is formed out of classes belonging to the ASDF package. The largest classes from this package are Operation and Component. The reason for the conglomerate of classes is that there are a series of generic functions that have methods for each combination of operation and component. This is a good example of double-dispatch use. In Java or C++, this problem would be solved by applying the Visitor pattern. But in CLOS, with the help of multiple-dispatch, we can resolve this problem much more easy and elegant by providing a method for each possible combination of operation and component. The other classes in the conglomerate are subclasses of the two above mentioned classes. They are in this conglomerate because there are also methods that specialize on subclasses of Operation and subclasses of Component. The conglomerate from the left of the view is made from classes belonging to the LISA package, containing the implementation of the production rule system. The explanation for this conglomerate is the same as the first case: multi-methods that specialize on different type of nodes, sub-classes of JOIN-NODE and SHARED-NODE, and different type of tokens, sub-classes of TOKEN. TOKEN represents the tokens from the rules that are used in the system in the process of pattern matching, while JOIN-NODE and SHARED-NODE represent two types of nodes from the search network, or graph.

The three large classes tied in a chain, from the center of Figure 7.8, represent the main entities of the system: the Rete algorithm (RETE class), rules (RULE class) and facts (FACT class). A production system, like the Rete algorithm, provides the mechanism necessary to execute productions, or rules, in order to achieve some goal for the system, represented by facts. The middle class from the chain is RETE, the class that implements the Rete algorithm. This class is connected with each of the other two classes by a multi-method, that specialize on the RETE class and on the RULE class, respectively FACT class. Because neither RULE or FACT don't have any subclasses to require a polymorphic call, we examined the two methods. We discovered that the reason for the polymorphic call is that a rule and a token can be represented by an RULE object, respectively TOKEN object, by an id number or by a string, and consequently there is an implementation method for each case. Also we observed that the RETE class doesn't have no subclasses, but the methods dispatch on its type. This choice is probably made for assuring the system's extensibility, by sub-classing the algorithm class with a possible extension to the algorithm or an improved version of it.

In Figure 7.9 we have the Class Types view, with scatterplot layout, of the Lisa project.

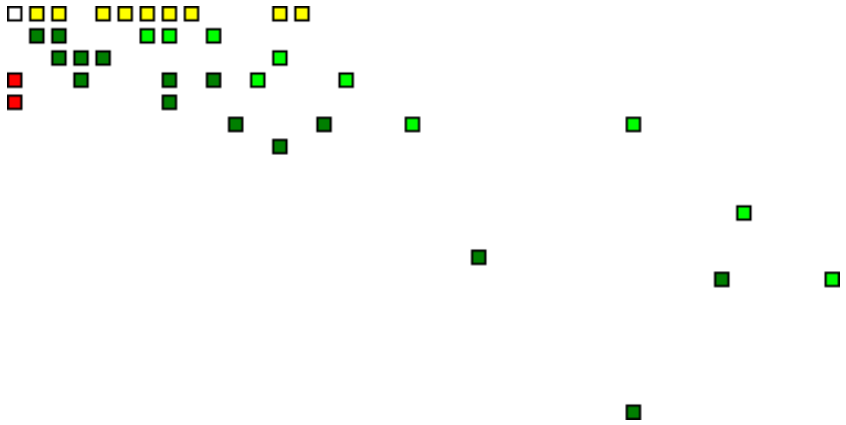

Figure 7.9: Lisa: Class Types View - scatterplot layout

When studying the distribution of class types from the Lisa project, in Figure 7.9, we observe that most of the classes have an average attribute to method ratio, with an small tendency to more methods, this showing a good class design. But there are also extreme classes, like the yellow ones, with only methods, or the red ones, with only methods. We also observe that the largest classes, those in the bottom left side, are green classes, which means they have an optimal attribute to methods ratio.

In Figure 7.10 we have the Class Types view, with tree layout, of the Lisa project.

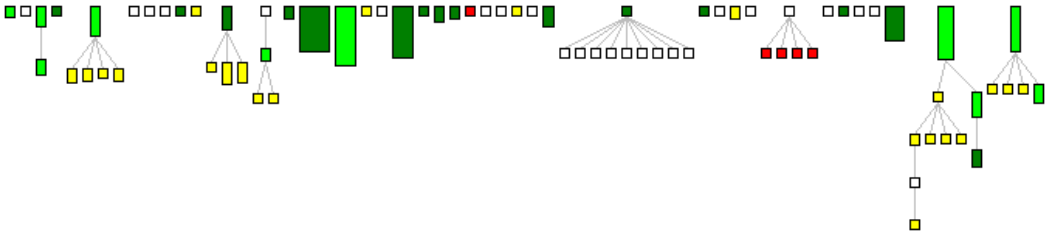

Figure 7.10: Lisa: Class Types View - tree layout

When we take a look at the Class Types view based on the tree layout we observe that almost all the green classes are on the top of the hierarchies. On the lower level of the hierarchies most of the classes are yellow, meaning that they only provide new functionality to the base class and no new state variables. One exception is the small inheritance tree that has red class on the second level. On inspection of those classes we found that they are a test case for the algorithm (the "monkey and bananas" planning problem) and they only provide the necessary test case information, rules and facts, without having any associated methods. A strange class tree in this view is the one with only white classes, empty classes, on the second level of the hierarchy,

meaning that those classes have no attribute and no associated methods. Upon inspection of those classes, we found that they are used in a test of Lisa's uncertainty mechanism as simple symbols, like Prolog atoms.

In Figure 7.11 we have the Generic Concerns view of the Lisa project. In this figure we can see a *scattered concern*, that is spread all over system's class hierarchy. The visualized generic concern is the *print-object* generic function, also presented in the SBCL case study.

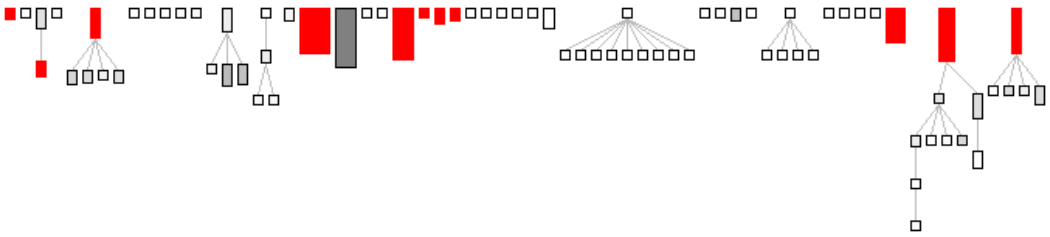

Figure 7.11: Lisa: Generic Concerns View

Being a relatively small project, only 77 classes, and having a simple, flat, inheritance hierarchy there's not to many chances of finding cross-cutting concerns in this project.

Chapter 8

Conclusions

Every end is a new beginning.

Software reverse engineering is a complex and difficult task, mainly because of the sheer size and complexity of software legacy systems. To be able to reason about software systems, software engineers need a meta-model to describes what and in which way information is modeled. The meta-model not only determines if the right information is available to perform the intended reengineering tasks, but also influences issues such as scalability, extensibility and information exchange.

In this thesis we developed the FAMIX-Lisp meta-model, as an extension of the FAMIX meta-model. Our FAMIX-Lisp meta-model extends the FAMIX meta-model with capabilities to model Lisp systems by adding new entities: Macros and CLOS entities. To test our meta-model we developed two tools that implement the FAMIX-Lisp meta-model: a model extractor (*ModeLisp*), that extract FAMIX-Lisp models from Lisp systems and a Lisp plugin (*MoosLi*) for the MOOSE environment, that can import FAMIX-Lisp models and browse the model and apply different analysis and visualizations on them. With the help of our tools we managed to model large systems, up to 350KLOC, obtaining models containing up to 70,000 entities, demonstrating the scalability of our meta-model and also of our tools.

Among the various approaches to support reverse engineering that have been proposed in the literature, software visualization became one of the major approaches. In our thesis we also presented a set of new visualizations for Lisp systems:

- The Class-Method Relation View: is a visual way of supporting the understanding of the relation between classes and methods in a object-oriented Lisp program;
- The Class Types View: is a visual way of identifying different types of classes, based on their structure (the attributes to methods ratio);
- The Programming Style Distribution View: is a visual way of identifying the programming paradigm used in a program and their distribution over the system's packages;
- The Generic Concerns View: is a visual way of identifying different cross-cutting concerns in a system by visualizing the spread of generic functions in the system.

The views presented in this thesis have been applied on several large Lisp projects, ranging in size 10KLOC to 350KLOC. Furthermore, we have illustrated our approach by applying different views and by thus reverse engineering two case studies. We have been able to understand different aspects of the case study, among which an overview of the application, a discussion on the used programming paradigms, understanding how classes and Lisp style methods are related and the impact of multi-methods, a discussion on different types of classes, the detection of cross-cutting concerns, as well as the detection of several places where an in-depth examination is needed.

Future Work

During our research we encountered a number of questions that we believe that are worth of further investigation in the future.

First off all, we are very interested in a more detailed study of Lisp macros. In our thesis, due to a lack of time, we limited the macro-expansion modeling capabilities of our tools to only invocations and accesses. Because a macro can be expanded to arbitrary Lisp code, the result of a macro-expansion can be any entity from the language. This makes macros significantly harder to design and debug than normal Lisp code, and are normally considered a topic for the advanced Lisp developer. We would like to do a thorough study of the impact of macros on software development and on software comprehension, what in Lisp slang is called "macrology"(the art and science involved in comprehending a complex macro). Because the process of macro-expansion is done at compile time, we have to also support pseudo-dynamic analysis of Lisp systems, to extract all the macro-expansions that happen in a Lisp program at the compilation stage.

Because a purely static perspective of a program overlooks valuable semantic knowledge of a system, a dynamic analysis of Lisp systems would be valuable. Analyzing the properties of a Lisp running program would be especially interesting because of the highly dynamic nature of Lisp.

Another possible issue is software metrics. The multitude of existing metrics can be applied only in part to Lisp code. For example we propose in our thesis the number of S-Expressions as an size/complexity metric for Lisp code. But there's a need for more complex metrics to be developed, not only for source code measurements but also for design quality.

A final direction we would like to mention is analyzing the version history to understand the evolution of a Lisp project over time and to see if developing Lisp applications is different form other languages.

List of Figures

2.1 The reengineering life-cycle . . . 6
2.2 Conception of the FAMIX meta-model . . . 10
2.3 The core of the FAMIX meta-model . . . 11
2.4 Basic elements of a polymetric view . . . 15
2.5 A complexity view of a Java project . . . 16
2.6 The decomposition of a class blueprint into layers . . . 17
2.7 A blueprint visualization of a Java class . . . 17
2.8 A SHriMP view showing the architecture of the SHriMP program itself. . . . 18

4.1 The FAMIX Meta-model: before and after . . . 34
4.2 The FAMIX-Lisp Meta-model: Abstract part . . . 41
4.3 The FAMIX-Lisp Meta-model . . . 42
4.4 The FAMIX-Lisp Meta-model: the Lisp Core . . . 43
4.5 The FAMIX-Lisp Meta-model: the CLOS Core . . . 44

5.1 Complexity view of a single-inheritance Lisp system . . . 50
5.2 Complexity view of a multi-inheritance Lisp system . . . 51
5.3 A blueprint visualization of a Lisp class . . . 51
5.4 A Class-Method Relation View . . . 52
5.5 A Class-Method Relation View: classical class . . . 53
5.6 A Class-Method Relation View: lonely classes and methods . . . 54
5.7 A Class-Method Relation View: class-method conglomerate . . . 54
5.8 A Class Types View: scatterplot layout . . . 55
5.9 A Class Types View: tree layout . . . 56
5.10 A Programming Style Distribution View: example 1 . . . 58
5.11 A Programming Style Distribution View: example 2 . . . 59
5.12 A Generic Concerns View: tree layout . . . 60
5.13 A Generic Concerns View: package distribution map . . . 61
5.14 A Generic Concerns View: package distribution map . . . 62
5.15 A Generic Concerns View: tree layout . . . 62

6.1 The Moose architecture, including the MoosLi plugin . . . 64
6.2 The MOOSE environment with MoosLi plugin . . . 66
6.3 A Class-Generic Functions visual browser . . . 67

6.4 A Class-Method visual browser 67

7.1 SBCL: Programming Style Distribution View 72
7.2 SBCL: Class-Method Relation View 73
7.3 SBCL: Class Types View - scatterplot layout 74
7.4 SBCL: Class Types View - tree layout 74
7.5 SBCL: Generic Concerns Views: scattered concern 75
7.6 SBCL: Generic Concerns Views: localized concerns 76
7.7 Lisa: Programming Style Distribution View 78
7.8 Lisa: Class-Method Relation View 79
7.9 Lisa: Class Types View - scatterplot layout 80
7.10 Lisa: Class Types View - tree layout 80
7.11 Lisa: Generic Concerns View 81

List of Tables

4.1 The Macro entity . . . 35
4.2 The MacroExpansion entity . . . 36
4.3 The AbstractObject entity . . . 37
4.4 The GenericFunction entity . . . 38
4.5 The CLOSMethod entity . . . 39
4.6 The Class entity . . . 40
4.7 The Attribute entity . . . 40
4.8 The Namespace entity . . . 41

7.1 Case studies statistics . . . 69
7.2 Case studies . . . 70
7.3 SBCL project statistics . . . 71
7.4 Lisa project statistics . . . 77

Bibliography

[BDW99] Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.

[BG97] Berndt Bellay and Harald Gall. A comparison of four reverse engineering tools. In *Proceedings of WCRE (Working Conference on Reverse Engineering)*, pages 2–11. IEEE Computer Society Press: Los Alamitos CA, 1997.

[BG01] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the OMG/MDA framework. In *Proceedings Automated Software Engineering (ASE 2001)*, pages 273–282, Los Alamitos CA, 2001. IEEE Computer Society.

[BGW93] D.G. Bobrow, R.P. Gabriel, and J.L. White. CLOS in context — the shape of the design. In A. Paepcke, editor, *Object-Oriented Programming: the CLOS perspective*, pages 29–61. MIT Press, 1993.

[Cas98] Eduardo Casais. Re-engineering object-oriented legacy systems. *Journal of Object-Oriented Programming*, 10(8):45–52, January 1998.

[CCI90] Elliot Chikofsky and James Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[CEK+00] Jörg Czeranski, Thomas Eisenbarth, Holger M. Kienle, Rainer Koschke, Erhard Plödereder, Daniel Simon, Yan Zhang, Jean-François Girard, and Martin Würthner. Data exchange in Bauhaus. In *Proceedings WCRE '00*. IEEE Computer Society Press, November 2000.

[CMS99] Stuart K. Card, Jock D. Mackinlay, and Ben Shneiderman, editors. *Readings in Information Visualization — Using Vision to Think*. Morgan Kaufmann, 1999.

[DDN02] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[DDT99] Serge Demeyer, Stéphane Ducasse, and Sander Tichelaar. Why unified is not universal. UML shortcomings for coping with round-trip engineering. In Bernhard Rumpe, editor, *Proceedings UML '99 (The Second International Conference on The Unified Modeling Language)*, volume 1723 of *LNCS*, pages 630–644, Kaiserslautern, Germany, October 1999. Springer-Verlag.

[DG87] Linda G. DeMichiel and Richard P. Gabriel. The Common Lisp object system: An overview. In J. Bézivin, J-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings ECOOP '87*, volume 276 of *LNCS*, pages 151–170, Paris, France, June 1987. Springer-Verlag.

[DGK06] Stéphane Ducasse, Tudor Gîrba, and Adrian Kuhn. Distribution map. In *Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM '06)*, pages 203–212, Los Alamitos CA, 2006. IEEE Computer Society.

[DL05] Stéphane Ducasse and Michele Lanza. The class blueprint: Visually supporting the understanding of classes. *Transactions on Software Engineering (TSE)*, 31(1):75–90, January 2005.

[DTD01] Serge Demeyer, Sander Tichelaar, and Stéphane Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

[Duc97] Stéphane Ducasse. *Intégration réflexive de dépendances dans un modèle à classes.* PhD thesis, Université de Nice-Sophia Antipolis, January 1997. Thèse de l'Université de Nice-Sophia Antipolis.

[EBN02] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of c preprocessor use. *IEEE Trans. Softw. Eng.*, 28(12):1146–1170, 2002.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 1999.

[Fod91] John Foderaro. Lisp: introduction. *Commun. ACM*, 34(9):27, 1991.

[FP96] Norman Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach.* International Thomson Computer Press, London, UK, second edition, 1996.

[GÔ5] Tudor Gîrba. *Modeling History to Understand Software Evolution.* PhD thesis, University of Berne, Berne, November 2005.

[Gra93] Paul Graham. *On Lisp.* Prentice Hall, 1993.

[Gra04] Paul Graham. *Hackers & Painters.* O'Reilly, 2004.

[Gre07] Orla Greevy. *Enriching Reverse Engineering with Feature Analysis.* PhD thesis, University of Berne, May 2007.

[Gro04] Object Management Group. Meta object facility (MOF) 2.0 core final adopted specification. Technical report, Object Management Group, 2004.

[HHL+00] Ahmed Hassan, Ric Holt, Bruno Lague, Sebastien Lapierre, and Charles Leduc. E/R Schema for the Datrix C/C++/Java Exchange Format. In *Proceedings of WCRE (Working Conference on Reverse Engineering), Exchange Formats Workshop*, pages 284–286, Los Alamitos CA, November 2000. IEEE Computer Society Press.

[HS96] Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity.* Prentice-Hall, 1996.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[Kic96] Gregor Kiczales. Aspect-oriented programming: A position paper from the Xerox PARC aspect-oriented programming project. In Max Muehlhauser, editor, *Special Issues in Object-Oriented Programming.* Dpunkt Verlag, 1996.

[Kon97] Kostas Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In Ira Baxter, Alex Quilici, and Chris Verhoef, editors, *Proceedings Fourth Working Conference on Reverse Engineering*, pages 44–54. IEEE Computer Society, 1997.

[Kos03] Rainer Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance and Evolution: Research and Practice*, 15(2):87–109, 2003.

[Lan03] Michele Lanza. *Object-Oriented Reverse Engineering — Coarse-grained, Fine-grained, and Evolutionary Software Visualization.* PhD thesis, University of Berne, May 2003.

[LB85] Manny Lehman and Les Belady. *Program Evolution: Processes of Software Change.* London Academic Press, London, 1985.

[LD03] Michele Lanza and Stéphane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, September 2003.

[LDGP05] Michele Lanza, Stéphane Ducasse, Harald Gall, and Martin Pinzger. Codecrawler — an information visualization tool for program comprehension. In *Proceedings of ICSE 2005 (27th IEEE International Conference on Software Engineering)*, pages 672–673. ACM Press, 2005.

[Leh96] Manny Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124, Berlin, 1996. Springer.

[lis] The Lisa Project. http://lisa.sourceforge.net/.

[LS94] P. Livadas and D. Small. Understanding code containing preprocessor constructs. *IEEE Third Workshop on Program Comprehension*, pages 89–97, 1994.

[LS99] Panagiotis K. Linos and Stephen R. Schach. Comprehending multilanguage and multiparadigm software. In *Proceedings of the short papers of ICSM '99*, pages 25–28, August 1999.

[MADSM01] C. Best M.-A. D. Storey and J. Michaud. SHriMP Views: An interactive and customizable environment for software exploration. In *Proceedings of International Workshop on Program Comprehension (IWPC '2001)*, 2001.

[Mar98] Radu Marinescu. Using object-oriented metrics for automatic design flaws in large scale systems. In Serge Demeyer and Jan Bosch, editors, *Object-Oriented Technology (ECOOP '98 Workshop Reader)*, volume 1543 of *LNCS*, pages 252–253. Springer-Verlag, 1998.

[McC60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *CACM*, 3(4):184–195, April 1960.

[MGL06] Michael Meyer, Tudor Gîrba, and Mircea Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis 2006)*, pages 135–144, New York, NY, USA, 2006. ACM Press.

[MMM+05] Cristina Marinescu, Radu Marinescu, Petru Florin Mihancea, Daniel Ratiu, and Richard Wettel. iPlasma: An integrated platform for quality assessment of object-oriented design. In *ICSM (Industrial and Tool Volume)*, pages 77–80, 2005.

[Moo86] David A. Moon. Object-oriented programming with Flavors. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 1–8, November 1986.

[Mül86] Hausi A. Müller. *Rigi — A Model for Software System Construction, Integration, and Evaluation based on Module Interface Specifications*. PhD thesis, Rice University, 1986.

[NDG05] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE 2005)*, pages 1–10, New York NY, 2005. ACM Press. Invited paper.

[Pae93] Andreas Paepcke. User-level language crafting. In *Object-Oriented Programming: the CLOS perspective*, pages 66–99. MIT Press, 1993.

[Par94] David Lorge Parnas. Software aging. In *Proceedings 16th International Conference on Software Engineering (ICSE '94)*, pages 279–287, Los Alamitos CA, 1994. IEEE Computer Society.

[PBS93] Blaine A. Price, Ronald M. Baecker, and Ian S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.

[Pit94] Kent M. Pitman. Accelerating Hindsight — Lisp as a Vehicle for Rapid Prototyping, 1994.

[RD99] Tamar Richner and Stéphane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In Hongji Yang and Lee White, editors, *Proceedings of 15th IEEE International Conference on Software Maintenance (ICSM'99)*, pages 13–22, Los Alamitos CA, September 1999. IEEE Computer Society Press.

[sbc] Steel Bank Common Lisp (SBCL). http://sbcl.sourceforge.net/.

[SBM+02] Margaret-Anne Storey, Casey Best, Jeff Michaud, Derek Rayside, Marin Litoiu, and Mark Musen. SHriMP views: an interactive environment for information visualization and navigation. In *CHI '02: CHI '02 extended abstracts on Human factors in computing systems*, pages 520–521, New York, NY, USA, 2002. ACM Press.

[SDBP98] John T. Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors. *Software Visualization — Programming as a Multimedia Experience*. The MIT Press, 1998.

[SFM99] Margaret-Anne D. Storey, F. David Fracchia, and Hausi A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Software Systems*, 44:171–185, 1999.

[SM95] Margaret-Anne D. Storey and Hausi A. Müller. Manipulating and documenting software structures using SHriMP Views. In *Proceedings of ICSM '95 (International Conference on Software Maintenance)*, pages 275–284. IEEE Computer Society Press, 1995.

[SMHP+04] Rational Software, Microsoft, Hewlett-Packard, Oracle, Sterling Software, MCI Systemhouse, Unisys, ICON Computing, IntelliCorp, i Logix, IBM, ObjecTime, Platinum Technology, Ptech, Taskon, Reich Technologies, and Softeam. *Unified Modeling Language (version 2.0)*. Rational Software Corporation, September 2004.

[SS00] Susan Elliott Sim and Margaret-Anne D. Storey. A structured demonstration of program comprehension tools. In *Proceedings of WCRE 2000*, pages 184–193, 2000.

[Ste90] Guy L. Steele. *Common Lisp The Language, 2nd Edition*. Digital Press, second edition, 1990.

[SWM97] Margaret-Anne D. Storey, Kenny Wong, and Hausi A. Müller. Rigi: A visualization environment for reverse engineering. In *ICSE*, pages 606–607, 1997.

[Tic01] Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Berne, December 2001.

[Tuf90] Edward R. Tufte. *Envisioning Information*. Graphics Press, 1990.

[Tuf97] Edward R. Tufte. *Visual Explanations*. Graphics Press, 1997.

[WH92] Norman Wilde and Ross Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, December 1992.

[WL07] Richard Wettel and Michele Lanza. Visualizing software systems as cities. In *Proceedings of VISSOFT 2007 (4th IEEE International Workshop on Visualizing Software For Understanding and Analysis)*, pages 92–99, 2007.