How Lisp Systems Look Different

In Proceedings of European Conference on Software Maintenance and Reengineering (CSMR 2008)

Adrian Dozsa

Politehnica University of Timişoara

Romania

adi.dozsa@gmail.com

Tudor Gîrba

University of Berne

Switzerland

girba@iam.unibe.ch

Radu Marinescu

Politehnica University of Timişoara

Romania

radum@cs.utt.ro

Abstract

Many reverse engineering approaches have been developed to analyze software systems written in different languages like C/C++ or Java. These approaches typically rely on a meta-model, that is either specific for the language at hand or language independent (e.g. UML). However, one language that was hardly addressed is Lisp. While at first sight it can be accommodated by current language independent meta-models, Lisp has some unique features (e.g. macros, CLOS entities) that are crucial for reverse engineering Lisp systems. In this paper we propose a suite of new visualizations that reveal the special traits of the Lisp language and thus help in understanding complex Lisp systems. To validate our approach we apply them on several large Lisp case studies, and summarize our experience in terms of a series of recurring visual patterns that we have detected.

Keywords: reverse engineering, visualization, Lisp

1. Introduction

Software reverse engineering is about creating high level views of a target system [2]. While in theory reverse engineering techniques might be language independent, in practice the special traits of the various programming languages must be take into account. This usually implies creating a model of the source code, and then performing analyses on it the resulted model. The model must be rich enough to capture the most important facts about the analyzed system. Over the last decades, various approaches have been developed for systems written in various languages like Java, C++ or COBOL. One language that was little addressed in the reverse engineering community is Lisp [22].

The need to address the understanding and maintenance of Lisp systems is emphasized by the fact that Lisp is cur-

rently used in a variety of domains, like bio-informatics (BioBike), data mining (PEPITe), knowledge-based engineering (Cycorp or Genworks), video games (Naughty Dog), flight scheduling (ITA Software), natural language processing (SRI International), CAD (ICAD or OneSpace), financial applications (American Express), web programming (Yahoo! Store or reddit.com), telecom (AT&T, British Telecom Labs or France Telecom R&D), electronic design automation (AMD or American Microsystems) or planning systems (NASA's Mars Pathfinder spacecraft mission) [16].

Why Lisp is Different. In spite of its almost fifty-year history, and of the fact that other programming languages borrowed concepts from it, Lisp still presents some unique traits, that are decisive for understanding Lisp systems.

Multi-Paradigm. Started as a functional language, today Lisp is truly a *multi-paradigm* programming language. It incorporates all major programming paradigms: functional, procedural, logic, object-oriented, and even newer paradigms like aspect-oriented.

Multiple-Dispatch. Even in the context of object-oriented languages, Lisp sets itself apart from other languages with different language entities and new ways of combining those entities. CLOS (Common Lisp Object System) [4] offers a *multiple dispatch* system which 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 class or another.

Special Entities. Methods in CLOS are grouped into *generic functions, i.e.,* collections 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. CLOS also introduces a new type of classes, *mixin classes* [19], behavioral classes that are used additively through multiple inheritance.

Macros. Another unique Lisp feature consists of its

---

[1]This paper makes use of colors. Please read a colored printed, or an electronic version of this paper.

macros. Macros are a powerful means to write custom programming language constructs beyond mere functions. While apparently similar to C macros, they are different. While C macros are specifications of simple string substitutions, Lisp macros are Lisp programs that generate other Lisp programs [12].

Visualizing Lisp Systems.   In this paper, we argue that to effectively understand Lisp systems, we need dedicated analyses that take into account Lisp specific constructs. However, because not much work has been dedicated to analyzing Lisp systems, we propose to explore them through a set of visualizations that capture these particularities and that provide an insight into how Lisp systems look different from those written in other languages. Based on the visualizations we report on patterns we encountered in several case studies. We see this work as a first step towards more detailed dedicated analyses.

With the help of the visualizations presented in this paper, we can see how Lisp systems make use of multiple programming paradigms, we can understand the relation between classes and methods, as defined in CLOS, we can see how studying generic functions can help us identify crosscutting concerns, and we can also identify different types of classes found in object-oriented Lisp systems.

Paper structure.   In the next section (Section 2) we define four visualizations that address the aforementioned Lisp specificities. For each visualization we describe a set of patterns that support the interpretation. In Section 3 we present an overview of an extensive case-study, and discuss in detail the findings in the largest Lisp system that we analyzed. The paper is finalized with a discussion on related work (Section 4) and the paper's conclusions and future work (Section 5).

2. Lisp Visualizations

As not many reverse engineering works tackled Lisp systems, we aimed to provide a set of initial visualizations [2] that capture the specifics of Lisp, and we use these visualizations as an initial instrument to explore the unique features of Lisp. We regard this work as an initial step towards more detailed analyses.

Starting from the traits that make Lisp different, identified in Section 1 (*i.e.,* multi-paradigm language, multiple dispatch, generic functions, mixin classes), we have created 4 visualizations dedicated to reveal each of this differences respectively: (1) PROGRAMMING STYLE DISTRIBUTION VIEW, (2) CLASS-METHODS RELATION VIEW,

[2]All our visualizations were developed using Mondrian [18], an engine for scripting interactive visualizations.

(3) GENERIC CONCERNS VIEW, and (4) CLASS TYPES VIEW. The rest of this section provides details for each of these visualizations.

2.1. The Programming Style Distribution View

The PROGRAMMING STYLE DISTRIBUTION VIEW, see Figure 1, is a visual way of identifying the paradigms used throughout the packages of a system.

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 view represents a variation of a *Distribution Map* [5] and it illustrates the correlation between a chosen concept and the structural modularization of a system.

For each package there is a large rectangle and within the rectangle, for each contained software artifact (*i.e.,* functions, classes, methods, global variables, macros) there is a square. The color of the squares refers to the paradigm used by these artifacts as shown in the table below:

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

The order of the large rectangles in a Distribution Map is based on a hierarchical clustering algorithm (average linkage) [23], that groups similar elements together. The goal is to group together the rectangles that are visually similar.
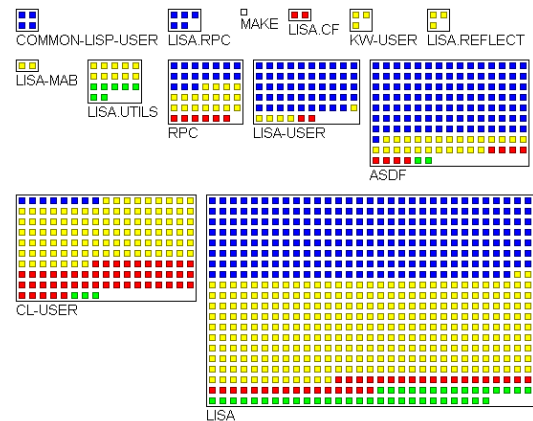

Figure 1. Programming Style Distribution View from the Lisa project

Reverse engineering goals. The PROGRAMMING STYLE DISTRIBUTION VIEW helps to identify the programming style(s) used in each package. This view is also useful in determining the size and complexity of the packages and the balance between the program's packages.

We applied this view on several Lisp systems (see Section 3) and by studying the outcome we have extracted a series of visual patterns, that can be grouped in the following categories: *size-based*, *structure-based* and *distribution-based* visual patterns:

Distribution-based patterns. These patterns are based on the distribution of the programming styles over the program's packages. We can distinguish two types of patterns. First we can have *distributed styles*, when the programming paradigms are more or less evenly distributed over all the packages in the system. Then we can have *encapsulated styles*, where each style of programming is encapsulated in one or a set of packages. The most common case is the distributed style. The separation of each programming style in different packages is rarely encountered. One style of programming style that makes an exception is the macro-based programming, that can be often found grouped in a set of packages. Usually, when a project makes heavy use of macros, it tends to group all the macros in one, or several packages. This contained group of macros will define a domain specific language, that then can be used in the project, for a greater abstraction of the code.

Structure-based patterns. Based on the structure of the packages, we can have the following patterns: *combined styles* (packages contain a mixture of styles) and *exclusive style* (packages contain in majority one programming style). Being a multi-paradigm programming language, the most common case in Lisp projects is that packages have a mixture of programming styles. We have also found some exceptions of packages that had used only one style of programming, mainly object-oriented programming.

Size-based patterns. This last category is based on the size of the packages and how this size is distributed throughout the system. First we can have *balanced packages*, where all the packages of the system are balanced in size, meaning that the program structure is well organized in packages. Then we can have *monolithic package*, where there is one big package compared to the rest of the packages, which contains most of the system. This characterizes a bad decomposition of the system in packages. Most of the analyzed projects had a balanced package distribution, with only a few larger packages than the average size. We ob-

served that the projects that are predominant object-oriented have a better package decomposition. But we have also found, in the case of older projects, some cases of monolithic packages, that contained most of the system.

2.2. The Class-Method Relation View

The CLASS-METHODS RELATION VIEW, see Figure 2, shows the relationships between classes and methods in a object-oriented Lisp application.

This view reveals the difference between CLOS and other object-oriented languages, like C++ or Java. As previously mentioned, in 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. Consequently, the relation between a class and a method is not one of containment but rather one of specialization.

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 the number of attributes for the class width and the number of corresponding methods for the class height.

Because each method can specialize on one or more classes, we can 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* [8]. This layout positions the nodes by resolving a system of forces. Nodes repel each other, while edges connecting them draw them together. The physical interpretation of this equilibrium state is the mechanical equilibrium. Because the layout will put together classes that have methods in common, we decided to map the number of methods both on nodes height and on connections to distinguish between the importance of classes when they are close to each other.

In Figure 2 we have an example for a small Lisp project with approximatively 40 classes. The view was obtained after a fixed number of iterations (300 in this case). The layout was not iterated until it reached the equilibrium state because that wouldn't have changed significantly the final layout.

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 they can study the overall system more easily. The components can be identified on the
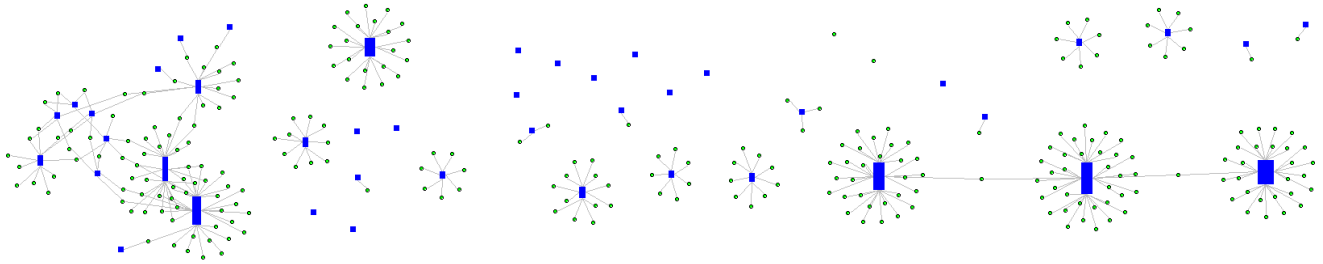
Figure 2. Class-Method Relation View, from the Lisa project

view by looking at conglomerates of classes and methods heavily connected, or by visually detecting loosely coupled components that have little or no connections to other parts of the system. Furthermore, the view can also help identifying classes that have no connection with any method or methods with no connection to any class.

From our case studies we have extracted a series of visual patterns, described below:

Stars. Star like figures with a blue rectangle in the middle, a class, and green circles around it. In this case these methods are connected to only one class, meaning that they specialize on only the class of one parameter. Such methods are like C++/Java methods which specialize on *this*, the class they belong to.

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 them. These classes are degenerated classes and we identified a series of causes for this cause: (i) the class is only used for data storage, similar to C's *struct*, and has no methods associated to it, also called Data Class; (ii) in projects that make heavy use of the macro feature, method definitions get "hidden" by macro definitions; (iii) they are *auxiliary classes*, that do not belong to the system but were introduced into the model for a better understanding of the system.

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 methods do not specialize on any class. This is possible in CLOS because here methods can 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*).Thus, *Lonely Methods* specialize on objects rather than classes, and because of that they are not connected to any class.

Conglomerates. Conglomerates are formed by a set of classes and methods heavily connected by edges and situated in a close vicinity to each other. These classes are pulled together by methods connected to more than one class, because a method pulls together the classes that it is connected to (*i.e.,* those classes it specializes on). As a result of these forces, the viewer sees a conglomerate of classes and methods, that lend the impression of entanglement. The methods that cause this conglomerates are called *multi-methods* [1], *i.e.,* methods that specialize on more than one class. This visual pattern is distinctive for CLOS systems, because in message-sending object-oriented languages there are no multi-methods.

2.3. The Generic Concerns View

Separation of concerns is a powerful principle that can be used to manage the inherent complexity of software [10]. One of the benefits of separation of concerns offers an increased understanding of how an application works, because the code belonging to a concern can be seen and reasoned about in isolation from the other concerns.

In "classical" object-oriented languages, where methods belong to classes, aspects can crosscut methods from different classes. Consequently, in AOP (Aspect-Oriented Programming), programmers are supposed to first define each of the system's aspects in isolation and then to let the defined aspects be automatically weaved together into the final code. In contrast to this, in generic-function systems, like Lisp/CLOS, the implementation of cross-cutting concerns is achieved easier because an aspect can be abstracted into a generic function, where this generic function implements aspect methods for each class.

We propose a means for identifying crosscutting concerns in CLOS applications by analyzing how the generic functions are spread over the system. We call these concerns, associated with generic functions, *generic concerns*. We define the GENERIC CONCERNS VIEW (see Figure 3) for visually representing how these generic concerns are spread over a system.
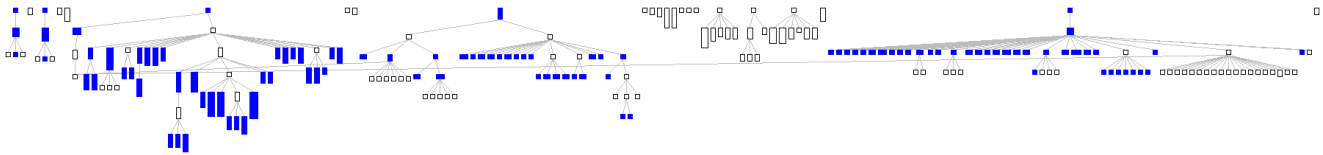
Figure 3. Generic Concerns View showing the `display-parse-tree` **concern from the Climacs project**

Description. The GENERIC CONCERNS VIEW (see Figure 3) is a polymetric view [13] and visualizes the impact of a generic function over the system's classes. The system is represented by its class inheritance hierarchy using a tree layout, whereby the nodes are classes, while the edges represent inheritance relationships. The metrics used in shaping each node are number of attributes for the width, and number of methods for the height of each node.

The key information of the view is given by how nodes are colored. Thus, if a node is colored in *blue* it means that the class has a corresponding method that is associated with the generic function for which the view is displayed (*i.e.,* the class has a corresponding method that has a class specific implementation of the generic function). In other words, *blue* colored classes are those that are dispatched upon by one of the methods of the generic function.

Because this visualization takes as input the generic function under study, it is best used in an interactive tool. Our prototype based on the Mondrian interactive visualization engine [18] allows us to select the generic function from a list and updates the view of the visualization. Thus, the tree layout of the classes forms the map on which the impact of the different concerns are highlighted.

Reverse engineering goals. As mentioned before, the GENERIC CONCERNS VIEW helps to identify and locate cross-cutting concerns associated with generic functions. By analyzing an extensive series of case-studies we have identified two recurring patterns for this view, representing two types of concerns:

Scattered concerns. Concerns that are scattered all over the inheritance hierarchy are usually general concerns that are common for most of the application's classes. For example, in Figure 3 we show an example of an `display-parse-tree` concern. Other examples of such concerns are instantiation, serialization, logging, drawing (in case of a graphical application), parsing (in a parser).

Localized concerns. These concerns are only spread in one subtree of the inheritance hierarchy, and they are closely related with the root of the subtree. For example, they appear in drawing (for a graphical object),

evaluation (for a mathematical expression), or event handling (for user interface components).

2.4. The Class Types View

The CLASS TYPES VIEW, see Figure 4, is a visual way of identifying different types of classes, based on their structure. Classes are classified based on the ratio between the number of attributes and number of methods. This characteristic of classes is visually encoded using colors.

There are at least 2 different types of classes in a Lisp object-oriented system, beyond the typical classes that we can find in C++/Java languages.

First, we can have *mixin classes*. A *mixin* is a class designed to be used additively, not independently, *i.e.,* is intended to be composed with other classes or mixins. The term *mixin*, or *mixin class*, was originally introduced in Flavors [19], a predecessor of CLOS. The difference between a regular, stand-alone class, and a mixin is that a mixin models some small functionality slice and is not intended for stand-alone use, but it is supposed to be composed with some other class needing this functionality. But not all classes that have only methods, and no attributes, are considered mixin classes. You can tell if a class is a mixin class by looking at its use. If it follows the previous mentioned rules, has only behavior and it is used additively through multiple inheritance, then it is considered to be a mixin. Consequently, mixin classes will appear higher in the class hierarchy.

Second, there is a special case of classes that we can encounter: *empty classes*. These are classes that neither have attributes nor any associated methods. There are three reasons why this type of classes exists: (i) they can be classes that are only used as symbols, and not in the sense of classes in the object-oriented paradigm; (ii) they are *auxiliary classes* that do not belong to the system but are introduced into the model of the system for a better understanding of the system; these are, for example, system classes (*e.g.,* Object or Class) that are inherited by an application class and therefore included in the model, and because they were introduced just as auxiliary classes they do not have any attributes or methods; (iii) they can also appear in projects that make heavy use of the macro feature, and
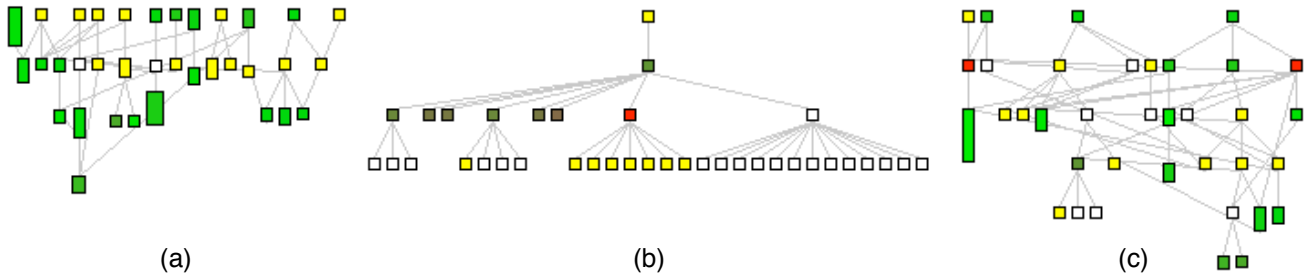
Figure 4. Class Types Views

where the class definition is "hidden" by macro definitions.

To identify the aforementioned special types of classes we need the following information: the number of attributes of a class, the number of methods of a class, and their position in the class hierarchy. Based on these requirements we have obtained the CLASS TYPES VIEW, defined as follows.

Description. The CLASS TYPES VIEW is also a polymetric view [13] and visualizes the system's class inheritance hierarchy with a tree layout. The metrics used to enrich this view are number of attributes (NOA) for the width and number of methods(NOM) for the height. The color of the nodes is based on the following coloring schema:

NOA = 0 and NOM = 0	white
NOA = 0	yellow
NOM = 0	red
others	light-green (more methods) to brown (more attributes)

In Figure 4 we have several examples of CLASS TYPES VIEWS, chosen to reveal the visual patterns that are presented in the following paragraph.

Reverse engineering goals. The CLASS TYPES VIEW helps to identify different types of classes, based on their structure, more precisely on the attributes to methods ratio. By studying the hierarchies of classes from several case studies, based on the previously mentioned coloring schema, we have identified a series of visual patterns:

Yellow root nodes and green inner-nodes and leaves (see Figure 4 case a): this represents the most common use of mixins, multiple inheritance via mixin composition. The root nodes are predominant yellow (the mixin classes), with some green nodes (representing the base class, that the mixins are combined with), while the inner nodes and leaf nodes are green (representing implementation classes).

Yellow leaves (see Figure 4 case b): in this case the leaf nodes are classes that only add new functionality to the super-class, without adding any new data. In this case, the yellow leaves do not represent mixins, because these classes are not a set of methods that will be use through inheritance by other classes, but these are subclasses that only have associated methods. This pattern is present quite often, and is in most cases cases related to the use of multi-methods. Based on two groups of these type of classes, we have a set of multi-methods that specialize on each combination of them. This case represents an multi-dispatch solution for the Visitor pattern [6], where one set corresponds to the visited classes and the other one to the visitors.

White leaves (see Figure 4 case b): in this case the leaf nodes are empty classes, with no data and no associated methods. This pattern usually indicates that these classes are used as symbols and not as normal object-oriented classes, *e.g.,* the symbols from an language (html tokens). Another cause of these white classes is the use of macros. The operations on these classes, the associated methods, are "hidden" behind a set of macro definitions, that form a small domain specific language, corresponding to the domain of the problem.

Inner red node (see Figure 4 case b or c): they represent classes that have only attributes. They usually are regarded as Data Classes, classes whose only purpose is to store data. We observed that this is valid only when they appear as leaf nodes. But when encountered as inner-nodes in the class hierarchy, they represent classes that have only state and the behavior is provided by its subclasses. This case is similar with *abstract classes* from C++/Java, which have state and virtual methods, that are then implemented by the subclasses. We call these inner red classes *Apparent Data Classes*.

3   Evaluation of the approach

In order to validate our approach, and to identify the recurring patterns for the newly defined visualizations, we applied them on several medium to large Lisp case studies. We have analyzed several open-source active Lisp projects, from small projects to large ones, up to 365 KLOC. The projects were chosen from a variety of domains: graphical libraries, artificial intelligence projects, musical applications, web servers, mathematical software, windowing systems, interface managers, editors, compilers.

In Table 1 we list some of the case studies we performed[1], with a series of additional information extracted from the models of the analyzed systems, like total number of lines (including comments and white lines), and number of structural entities (packages, classes, generic functions, methods, functions, macros, and global variables).

Project	LOC	Entities
SBCL	365314	7546
CL-HTTP	309006	17784
CLOCC	221216	8589
ACL2	219957	4686
Maxima	211630	9456
McClim	138891	13570
Closure	79500	7627
CommonMusic	59020	2861
GBBOpen	25158	674
Slime	21453	884
AlegroServe	16508	1045
Lisa	13443	1437

Table 1. Case studies

In the rest of this section, we present the findings in the most representative case study that we analyzed, namely the SBCL project, one of the largest open-source Lisp projects[2].

Steel Bank Common Lisp (SBCL) is an open-source compiler and runtime system for ANSI Common Lisp, derived from the CMUCL system[3]. SBCL is distinguished from CMUCL especially by a greater emphasis on maintainability. 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.

In Table 2 we summarize the most important size characteristics of the SBCL project. When we compare the numbers from this project with the average values from other

---

[1]http://www.cl-user.net/

[2]http://sbcl.sourceforge.net/

[3]http://www.cons.org/cmucl/

Number of lines	365314
Number of extracted entities	42787
Number of packages	54
Number of functions	4449
Number of macros	693
Number of classes	95
Number of generic functions	472
Number of methods	571
Number of methods per class	6
Number of attributes per class	1.2
Number of methods per generic function	1.2
Number of methods per attribute	5

Table 2. SBCL project statistics

Lisp projects, we notice that the density of structural entities per line of code is smaller than average, a typical situation for most of the large projects.

3.1   SBCL: Programming Style Distribution View

Looking at Figure 5 we first observe that we have two types of packages: some with one dominant programming style (*i.e.,* the *Exclusive Style* pattern), like SB-GRAY or SB!BIGNUM and to a lesser extent ASDF, and others that combine more programming styles (*i.e.,* the *Combined Styles* pattern), like SB-PCL, SB!C, SB!IMPL or SB!VM.
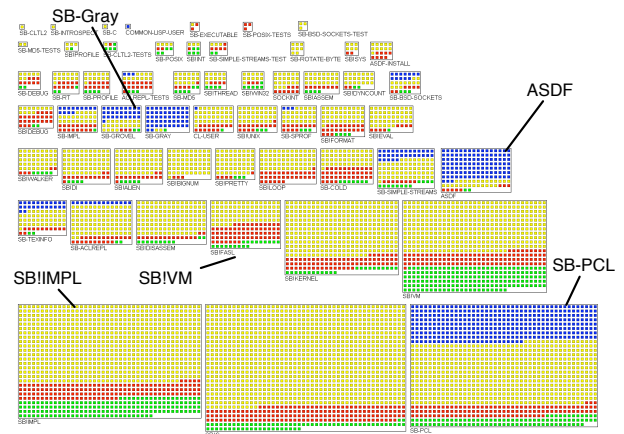

Figure 5. SBCL: Programming Style Distribution View

The object-oriented paradigm is used (the blue boxes) in several packages, but mainly in SB-PCL, ASDF and SB-GRAY. SB-PCL is the package responsible with the definition of PCL (Portable Common Loops) which is a portable CLOS implementation; this explains why this

package contains more object-oriented code. `ASDF` (Another System Definition Facility) is a system build utility (similar to *make*). It takes advantage of modern Common Lisp features and it uses CLOS for extensibility. Finally, `SB-GRAY` implements an extensible object-oriented version of Lisp streams. This was introduced from the inability to customize or extend stream behavior. In all these cases it is clear why this package are almost exclusively object-oriented.

Another interesting aspect consist of the distribution of macros over the packages (green boxes). We notice that the majority 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 virtual machine. The second package that contains a many macros is `SB!IMPL` that is responsible with the implementation of the language, the language definition. Being a metacircular language, the Lisp language definition is implemented in terms of Lisp macros.

3.2   SBCL: Class-Method Relation View

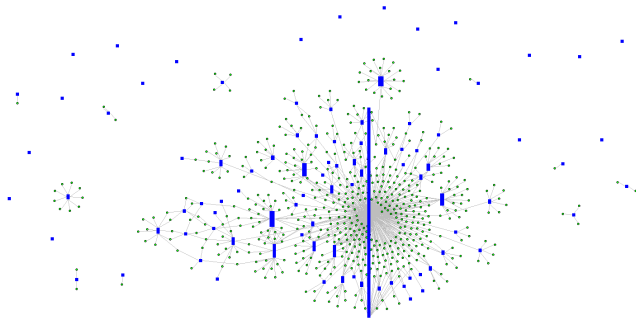An interesting result of applying this view to SBCL is depicted in Figure 6.

Figure 6. SBCL: Class-Method Relation View

We see a giant class in the center of the view and most of the other classes being attracted by 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* pattern. Obviously, 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. Other major classes are `OBJECT`, `CLASS`, `SLOT`, `GENERIC-FUNCTION` or `METHOD`, which are actually 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`.

3.3   SBCL: Generic Concerns View

In Figure 7 we show a *scattered concern*, that impacts all parts of the system's class hierarchy. The concern is present in all the sub-trees of he hierarchy, 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.
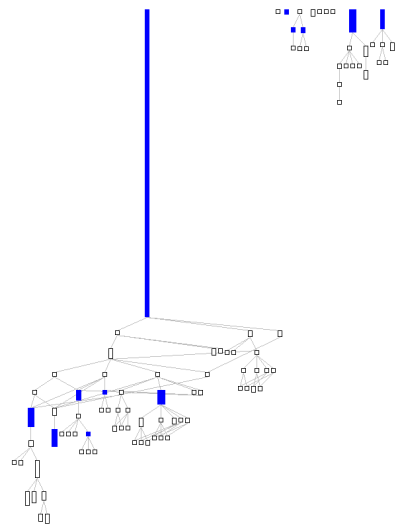

Figure 7. Scattered concern in SBCL

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.

3.4   SBCL: Class Types View

The CLASS TYPES VIEW puts again the giant `T` class in the center of attention (the very tall yellow bar in Figure 8). But we can see now that codeT is the root of the class hierarchy representing the implementation of the object system. Also note that the upper classes from the main tree are yellow, meaning that they have only functionality and no state at all. This does not mean that they are mixin classes, because they are not used additively with another base class, with the help of multiple inheritance.
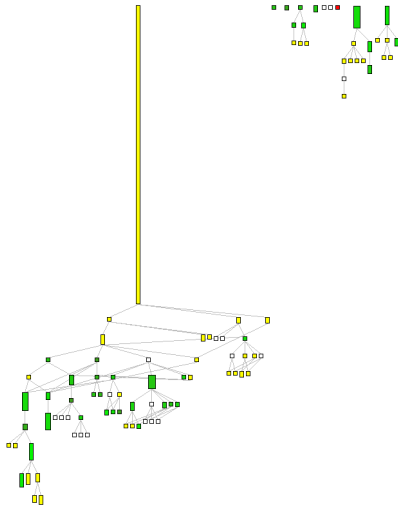
Figure 8. SBCL: Class Types View

When analyzed in more detail the main inheritance tree is very similar with the CLOS Meta-Object Protocol hierarchy [9]. These upper yellow classes are similar with Java interfaces, with the difference that in this case the methods belonging to these classes do have implementations, and not only define an interface to a class. The white leaves, that we observe on the bottom of the main tree, are some keywords used in the implementation of Lisp's object system from the compiler, and that are used as language keywords, but defined internally as classes. There is only one red class in the figure, *i.e.,* a class that only has state and no functionality. Upon inspection that class turned out to be a test class containing some test cases represented as attributes.

4   Related Work

The importance of software visualization as an approach to reverse engineering and program understanding can be inferred from the large number of tools and techniques that have been developed for this purpose: Rigi [20], SHriMP [17], CodeCrawler [14], Mondrian [18], etc. However, most of these tools focused on analyzing and visualizing mainstream programming languages, like C/C++ or Java. There are only a few visualization tools and techniques that address Lisp systems.

Haarslev and Möller [7] developed a framework for visualizing object-oriented systems written in CLOS, based on a TEX-like notation for specifying graphical layout of arbitrary objects. Using the framework, users can define layout descriptions declaratively, and apply them directly to the system under analysis. The meta-level architecture of CLOS is used to associate visualizations with application objects, requiring no source code modifications of applica-

tion systems. Some example uses of this framework are a class browser, generating a interactive graph of a class hierarchy, or a class inspector.

GraphTrace [11] is a tool for visualizing the dynamic behavior of object-oriented programs. GraphTrace allows a user to create displays revealing different aspects of the structure of an object-oriented program, and then to animate these displays in order to understand how the program works. The tool works with both structural and behavioral views of a system. Structural views are simply generated from the code's structure. Behavioral views are generated by recording the message activity that occurs in the execution of the program and then animating those views. Concurrent animation provides the user with several parallel views on a systems behavior. GraphTrace was developed in and for the Strobe language [21], an extension of CLOS.

The Scheme Macro Stepper [3] is a macro debugger with full support for modern Scheme macros. The macro debugger shows the macro expansion process as a reduction sequence, where the reducible expressions are macro applications and the contexts are primitive syntactic forms, i.e., nodes in the final abstract syntax tree. The debugger also includes a syntax display and browser that helps programmers visualize properties of syntax objects.

ZStep [15] is a program debugging environment designed to help the programmer understand the correspondence between static program code and dynamic program execution. Some of ZStep's features are: an animated view of program execution, complete history of program execution and output, "video recorder" controls to run the program in forward and reverse directions and control the level of detail displayed, direct access from graphical objects to the code represented, direct access from expressions in the code to their values and graphical output. ZStep is implemented in Macintosh Common Lisp, and it works with a subset of Common Lisp.

The novelty of our approach as compared to the above ones is that our visualizations represent the system on a higher abstraction level. All of the previous presented approaches, view the system under analysis at the code level. Our visualizations view the system at a higher level, viewing the system's entities (classes, methods or packages) and the relation between them and not its code.

5   Conclusions

In spite of optimistic claims, most reverse engineering and software visualization techniques have significant language dependent traits. So far, the main focus of reverse engineering techniques has been on mainstream object-oriented languages like C/C++ or Java. However, one language that was hardly addressed is Lisp, even though there exist a significant number of legacy Lisp systems. These

systems cannot be accommodated by current techniques as Lisp has a set of unique language features (*e.g.,* macros and CLOS entities).

In this paper we defined a suite of visualizations which take in consideration Lisp's unique features. Specifically, the visualizations support: (i) the exploration of the use and distribution of programming styles with a Lisp system; (ii) the understanding of the connection between classes and Lisp-style methods and the impact of multi-methods; (iii) the distinction between various types of Lisp classes; (iv) the detection of cross-cutting concerns defined by means of generic functions.

We have extracted a series of recurring visual patterns based on applying these visualizations on a significant number of projects. We see these patterns as an initial study of the specifics of Lisp systems. In the future, we plan to do a more detailed study of Lisp macros as these are significantly harder to design and debug than normal Lisp code.

References

[1] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. Commonloops: merging Lisp and object-oriented programming. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 17–29, New York, NY, USA, 1986. ACM.

[2] E. Chikofsky and J. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, Jan. 1990.

[3] R. Culpepper and M. Felleisen. A stepper for scheme macros. In *Scheme and Functional Programming*, 2006.

[4] L. G. DeMichiel and R. 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.

[5] S. Ducasse, T. Gîrba, and A. 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.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995.

[7] V. Haarslev and R. Möller. A framework for visualizing object-oriented systems. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 237–244. ACM Press, 1990.

[8] M. Kaufmann and D. Wagner, editors. *Drawing Graphs: Methods and Models*, volume 2025 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[9] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *Proceedings ECOOP '97*, volume 1241 of *LNCS*, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag.

[11] M. F. Kleyn and P. C. Gingrich. Graphtrace - understanding object-oriented systems using concurrently animated views. In *OOPSLA '88: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 191–205, New York, NY, USA, 1988. ACM Press.

[12] D. B. Lamkins. *Successful Lisp: How to Understand and Use Common Lisp*. bookfix.com, 2004.

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

[14] M. Lanza, S. Ducasse, H. Gall, and M. 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.

[15] H. Lieberman and C. Fry. Bridging the gulf between code and behavior in programming. In *CHI '95: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 480–486, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co.

[16] Association of Lisp Users Wiki Page - Industry Applications using Lisp. http://wiki.alu.org/Industry_ Application.

[17] C. B. 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.

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

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

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

[21] R. G. Smith. Strobe: support for structured object knowledge representation. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pages 855–858, 1983.

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

[23] R. Xu and I. Wunsch, D. Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, 16(3):645–678, June 2005.