Prawn: An Interactive Tool for Software Visualization

Andrew Chan, Reid Holmes

Department of Computer Science
University of British Columbia
201-2366 Main Mall
Vancouver BC Canada V6T 1Z4
{chana, rtholmes}@cs.ubc.ca

ABSTRACT

Software systems, by their very nature, are complex abstractions of the physical world. Due to this complexity, it is often extremely difficult for developers to fully comprehend the system they are working on, particularly when they were not involved in designing it. However, this comprehension is critical; over a system's lifespan, more time is spent maintaining it than in any other phase. We introduce the Prawn tool, which uses a variety of information visualization techniques to allow developers to see how their code interacts with the rest of the system. We believe Prawn reduces the time and effort required to make a change to a system, and increases developer's confidence in their changes.

1. INTRODUCTION

Box and arrow diagrams are a key tool in any software designer's toolkit. Nearly any system documentation will contain class diagrams of the system or parts of the system, whether in UML[1] or in another notation. These diagrams are much easier to understand than a textual description of the classes, and they can convey a lot of information in a standardized way.

When a developer joins a software project, the project is often relatively mature, and the developer's role is to help maintain or extend the code base to meet the customer's needs. As a result, the developer must learn about the system quickly, often on the fly, such as when he or she is asked to make a change. The primary source of support for the developer may not be the system designers or developers with expert knowledge of the system, but design diagrams and other documentation. The architects of the system may have left the company, and with the size and complexity of software systems, it is unlikely that any one person would understand every facet of the system.

Unfortunately, design documentation has its shortcomings. It may be incomplete because it was not maintained after the initial release of the system. The documentation may contain incorrect details, or the implementation may violate aspects of the design, either through error or due to issues that arose during implementation, such as performance problems. Even if the documentation is correct and up-to-date, it is likely to be quite voluminous for any non-trivial system.

To make a change, developers typically need detailed information about specific parts of the system, and a high-level understanding of how these parts interact with one another and the rest of the system. Although design documentation, even with its shortcomings, would be valuable, a useful tool for the developer would be an interactive application that presents the design of the system, based on the actual implementation. The interactive nature of the application would allow each developer to focus on the areas of interest to them.

An interactive system would also be useful for developers who are considered experts in the system. With the size of modern software systems, diagrams help expert developers reason about the system, and can assist them in explaining the implementation to new developers. Using an interactive system, the expert developer again would not have to attempt to find the correct design diagram, but instead could manipulate the view to suit his or her needs. An expert developer could also use this system to evaluate whether the implementation of a system conforms to its design, and to determine the feasibility of making changes to the system.

Software visualization is an active area of research. There are two broad categories of visualization: dynamic and static. Dynamic visualization allows a developer to see the behavior of a system over time, given a certain set of inputs. It may or may not reveal details about the system structure, but is useful for tasks such as performance tuning, or debugging.

Static visualization presents the structure of the system based on an analysis of its source code. The main problem has been enabling an approach to scale to large systems. Two static visualization tools are the Rigi tool [7] and the Simple Hierarchical Multi-Perspective (SHriMP) tool [13]. Rigi provides a semi-automated process to reverse engineer a system's design from its source code. SHriMP builds on this, providing an alternate means of visualizing large data sets (such as software structure) using a zoomable user-interface and multiple views.

We were motivated to examine this area for several reasons. One of the authors hoped to use a software visualization tool to see whether the implementation of a software system

[1]`http://www.omg.org/uml/`

conformed to its design. We were not satisfied with the ones we tried; either the tool did not contain the functionality we desired, or it was too difficult to use. At the same time, we felt that with recent advances in software source analysis tools[2], and zoomable user interface toolkits[3][4], it would be possible to concentrate on the visualization aspects of a tool, rather than having to build the infrastructure ourselves.

In this paper, we describe the Prawn tool. It is a static visualization tool designed to assist a developer in understanding the structure of a Java system. Although it would be most useful to a developer who has domain knowledge of the system being analyzed, it could also help a developer explore an unfamiliar system. Usability and utility were high priorities in creating Prawn; we felt that a developer should be able to use the tool with minimal effort and training.

The rest of this paper is as follows: In Section 2, we describe the key features of Prawn. Following this, in Section 3 we present three sample scenarios that illustrate how Prawn could be used. Section 4 describes the related work in this area, with a particular comparison to the SHriMP tool. Section 5 presents an evaluation of the strengths and weaknesses of Prawn, while Section 6 presents future work which would increase the utility of the Prawn system. The final section concludes.

2. PRAWN

Prawn allows a developer to view the structure of any Java system easily. The developer only needs to specify as a command-line argument the name of a JAR file containing the classes composing the system. Prawn analyzes the system and displays the windows shown in Figure 1.

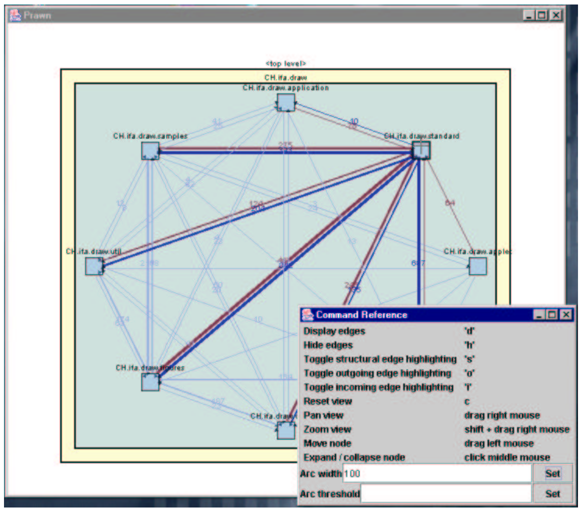
The view presented to the developer mimics the appearance of a UML class diagram with a node and arc graph. This was done to help reduce the learning curve associated with using the tool, and thus increase the chances that actual developers would use it. Nodes represent package, abstract class, class, or interface constructs in the system. Arcs represent calls between constructs, and an estimate of the number of calls made between constructs is provided[5]. Many information visualization techniques were used to ensure that the visualization would be as useful as possible.

2.1 Colour

A developer can easily distinguish between the different kinds of nodes by their colour: packages are blue, abstract classes are dark red, classes are light red, and interfaces are yellow. To reinforce the notion of packages containing subpackages, package nodes are semi-transparent so that packages that are several levels deep in the structure of the system are coloured a darker hue of blue. Colour is also used to distinguish between different types of arcs.

By default, arcs are shown in a muted blue colour so that they do not overwhelm the display. Arcs that pass through packages are visible due to the package transparency, but arcs that pass through classes are not. When a developer moves the pointer over a node, all of its arcs are displayed and highlighted, regardless of any other settings that may have been set. Incoming calls are shown in purple, outgoing calls are shown in blue, and structural calls (calls to a superclass, or an interface that's implemented, for example) are shown in green.

Although the way colour is used in Prawn does not take advantage of the fact that it can be perceived preattentively, we believe that a developer would primarily be interested in systematically examining the visualization, rather than looking for things to pop-out.



Display edges	'd'
Hide edges	'h'
Toggle structural edge highlighting	's'
Toggle outgoing edge highlighting	'o'
Toggle incoming edge highlighting	'i'
Reset view	c
Pan view	drag right mouse
Zoom view	shift + drag right mouse
Move node	drag left mouse
Expand / collapse node	click middle mouse

Arc width 100

Arc threshold

Figure 1: Prawn Canvas and Command Window

2.2 Aggregation

Any non-trivial Java system is composed of hundreds or thousands of classes organized into a package hierarchy. However, a developer is usually only interested in a subset of these classes. Within this subset, it is likely that the developer is especially concerned with a small number of classes; he or she needs to understand the rest at a high level. Prawn supports this by aggregating data along the package hierarchy, then allowing the developer to arbitrarily view parts of the system at different levels of detail.

The initial view that Prawn presents is of the *top-level* packages in the system. We define this as those packages located at the point where the package hierarchy in a system first diverges. For example, if the packages in a system include `ca`, `ca.ubc`, `ca.ubc.foo`, and `ca.ubc.bar`, the top-level packages would be `ca.ubc.foo` and `ca.ubc.bar`. If the system contains packages that differ immediately at the root level, these packages (say, `ca` and `org`) would be shown.

The developer can then select a package to view in more detail. The node representing the package is expanded in size and its contents are displayed within it. When nodes are expanded and nodes within nodes are expanded, this visually reinforces the nesting of classes and packages within the

[2]`http://www.alphaworks.ibm.com/tech/jikesbt/`
[3]`http://www.cs.umd.edu/hcil/jazz/`
[4]`http://zvtm.sourceforge.net/`
[5]Since Prawn uses static analysis of code, it is impossible to determine the exact number of calls. Method calls may be polymorphic, or may be made based on information that is provided at run-time

package hierarchy. This is shown in Figure 3. All visualization takes place within a single window; a multiple-window system would be difficult technically to keep synchronized, and consume valuable screen real-estate.

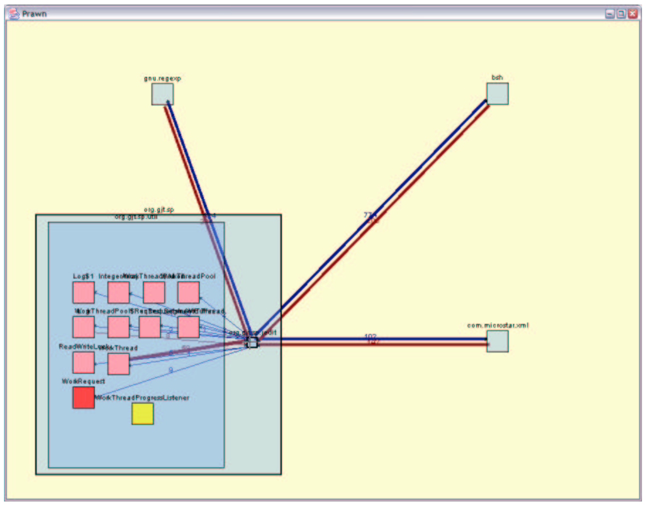


Figure 3: Prawn Canvas with Nested jEdit Packages

2.3 Navigation and Zooming

Once the developer has opened several packages to examine their contents, the nodes may not fit within the Prawn window. The developer can do one of three things:

- Pan over the contents
- Move nodes as needed so that the nodes of interest fit within a window. Arcs are automatically moved with nodes.
- Zoom in and out of the contents. This permits more content to fit within the window; text is scaled so that it remains readable no matter how far out the developer zooms.

Panning and zooming are two separate operations in Prawn, although they could have been combined, as shown in [6]. After experimenting with Jazz, a zoomable user-interface toolkit that implements this metaphor, we decided against using it, as we felt that developers would be able to grasp the separate concepts, but would struggle with controlling pan and zoom simultaneously.

Besides the zoom described, we also adopted a focus+context metaphor based loosely on fish-eye views. Nodes that are expanded occupy more screen real-estate than those that are not, although we did not have time to implement an algorithm to reclaim space when nodes are collapsed. Rather than having a single fish-eye lens that a developer would move over the visualization, we felt that developers would want to view multiple aspects of the system in varying levels of detail, simultaneously. This could be useful when a developer is trying to see the interaction between two particular classes that are in different packages, as shown in Figure 5.

2.4 Spatial Layout

The organization of nodes in the graph greatly affects its usability. Although a developer could use the functionality provided to organize nodes as he or she wished, it is time consuming to do so. As a result, in Prawn we used three different approaches to lay out nodes in a manner that we felt would be useful.

At first, we employed the straightforward strategy of laying out the nodes in a grid. Later, we noted that the number of times arcs passed through nodes en route to their destination node would be greatly reduced if nodes were laid out in a radial manner. At the same time, we wanted to have *closely related* nodes placed close to one another: a class should be placed close to its superclass, a class should be placed close to the interfaces it implements, and subclasses of a common superclass should be placed together. Moreover, the appearance should be similar to that of a UML diagram.

We decided to use each of these methods where appropriate. When laying out top-level package nodes, we employed the radial method. For inheritance and interface implementation hierarchies, we used the strategy of placing closely-related nodes together; this came to be known as the tree layout. Other nodes were laid out using the grid method.

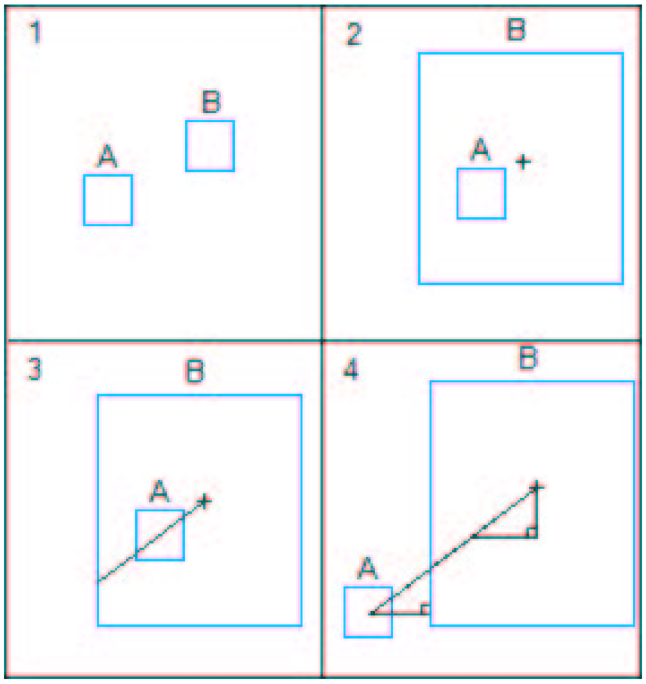


Figure 4: Occlusion Avoidance Algorithm

When a package is expanded to view its contents, nearby nodes often are occluded by the expanded package. A strategy was devised whereby occluded nodes were moved to the same location relative to the expanded node as before the node had been expanded. This is diagrammed in Figure 4. At first, two packages, A and B, are shown. Then package B is expanded; its new size is large enough that it will occlude package A. The new position of A is calculated by first determining the length of the line from the center of B through

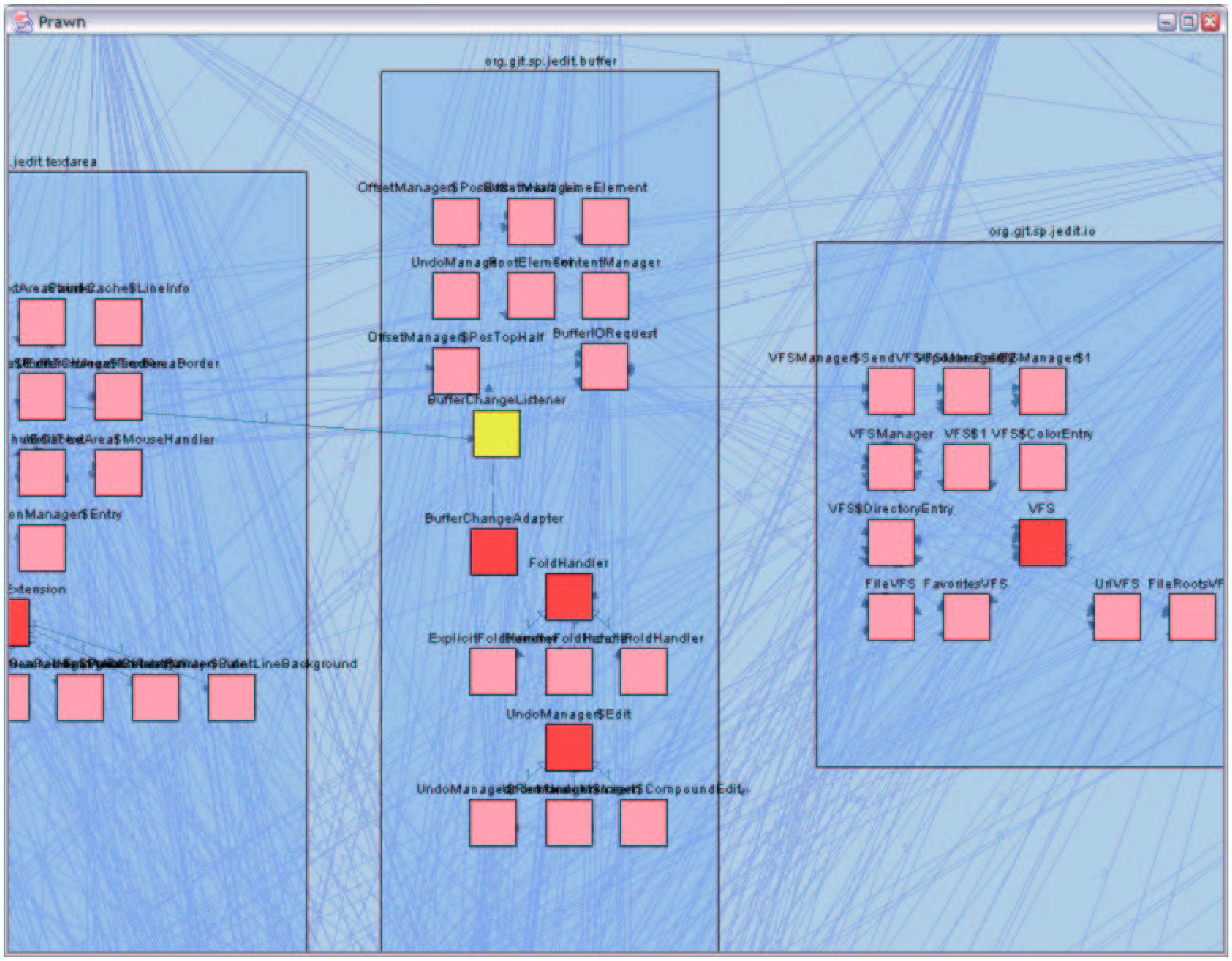


Figure 2: Prawn Canvas with Three jEdit Packages Open

the center of A to the edge of A. Originally, package A was moved by this amount, but to improve the appearance of the layout, this was changed to a percentage of the line length.

Another consequence of expanding a package is that the parent node often will not be large enough to contain the newly expanded package. When this happens, Prawn resizes nodes as necessary, from the parent node up to the top-level nodes.

2.5 Arc Management

Even with the varying level of detail and layout techniques discussed, the number of arcs shown in a graph quickly becomes unmanageable. As in the case of classes, a developer is typically only interested in a small subset of the calls made.

Prawn provides a number of techniques to help developers find arcs of interest and hide those that are not. These techniques work at different levels of granularity; some apply to the entire graph, while others only apply to a single node.

If the developer is interested in looking at classes that receive a large number of calls, he or she can identify these by specifying a value for an *Arc Width*. If an Arc Width of 50 is specified, an arc representing 1 to 49 calls will be a pixel wide, an arc representing 50 to 99 calls will be two pixels wide, and so on. Arcs can be up to four pixels wide. Another option for the developer would be to set an *Arc Threshold*. This is a value that defines the cutoff point for displaying arcs; if the arc represents fewer calls than the value, than it is only displayed when the developer moves the pointer over the node. Structural calls are not subject to this threshold.

At a finer-grained level, Prawn allows a developer to display or hide the arcs from a package, class, or interface to other nodes. One application of this would be to ignore classes that are used for debugging purposes only. While a developer is pointing at a given node, he or she can choose to keep incoming, outgoing, or structural calls highlighted after the pointer has been moved off. This would be useful when focusing on a given class. Figure 5 shows a view of Prawn with some of the arcs hidden.

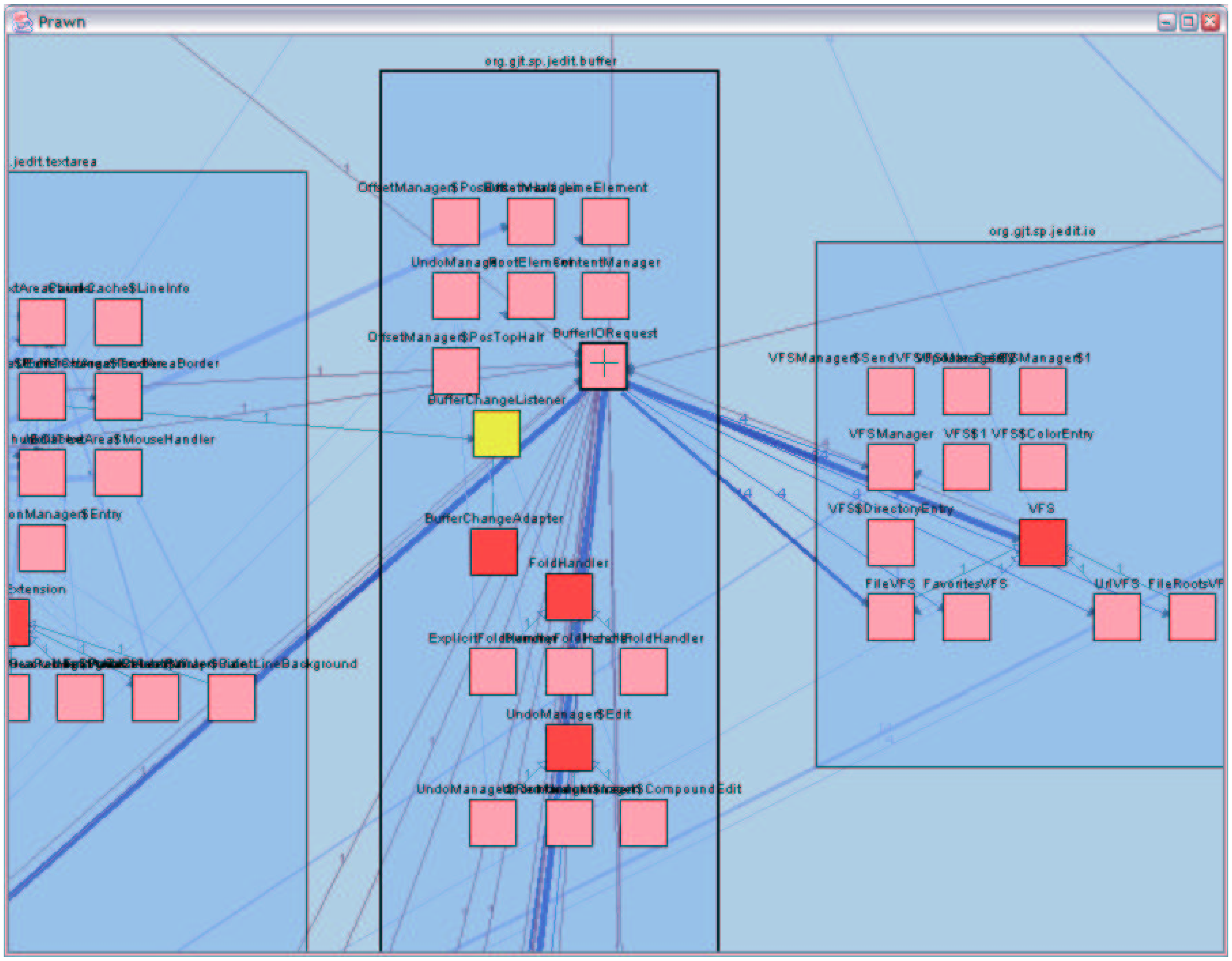


Figure 5: Prawn Canvas of jEdit with Many Arcs Elided

2.6 Implementation

Prawn was implemented using Java 2 and consists of several modules. The engine for analyzing Java classes was taken from the FEAT tool[6] written by Martin Robillard, a member of the Software Practices Lab at the University of British Columbia[7]. FEAT in turn uses the open-source JikesBT toolkit created by IBM for analyzing Java bytecode. A user interface module queries this engine to create an object representation of the nodes and arcs in the visualization. The actual visualization of the graph uses an open-source Java 2 zoomable UI toolkit called ZVTM.

Prawn is designed to be as extensible as possible, while maintaining the scope of visualizing software systems. It is designed so that a different Java analysis engine could be used instead of FEAT with minimal changes to the user interface module. As well, improved layout managers can be added by implementing an interface and indicating when the new manager should be used.

[6]`http://www.cs.ubc.ca/~mrobilla/feat/`
[7]`http://www.cs.ubc.ca/labs/spl`

Approximately 19 000 lines of commented Java code were written to create Prawn; approximately 8500 lines were slightly modified FEAT code. An issue that arose early in development was that FEAT would only return accurate results if a particular Java rt.jar was used - specifically, the version 1.4.1_01 JAR from one of the author's personal computer. Using earlier or newer versions of the JAR, or even downloading and installing a clean copy of the Java 1.4.1_01 JRE did not work. We suspect that the problem lies with the JikesBT toolkit, as it was originally designed to analyze Java 1.1.8 and Java 1.2.2 classes, but we were unable to verify this.

3. SAMPLE SCENARIOS

While Prawn is not suitable for all software evolution tasks, we present three scenarios where we feel that it would be useful.

3.1 Discovering Design Violations

One of the authors has worked on a medium-sized system that provides tools for Undergraduate Advisors in the Department of Computer Science. The CSSIS system consists of approximately 34 000 lines of code across 150 classes, and is subdivided into several modules. There are two major modules, one for advising and one for reporting. The author had designed the modules such that they would rely on common shared modules, but should not interact directly with one another. Prawn was used to test whether the implementation reflected this.

At the top-level, Prawn shows a `ca.ubc.cs` package. Expanding this package shows two packages: `cssis` and `sis`. Expanding the `cssis` package shows its 12 children, a mixture of packages, classes, and interfaces. Among the packages are the `advising` and `reporting` packages, which represent the advising and reporting modules.

Placing the pointer over the `advising` package, it was clear that there indeed were calls between the `advising` package and the `reporting` package in both directions. After expanding each of these nodes, the number of arcs shown on the screen became prohibitive. We chose to focus on calls from classes in the `advising` package to classes in the `reporting` package, so we hid all calls from packages besides these packages, then highlighted the incoming calls to the classes in the `reporting` package. This allowed us to easily identify the classes that would have to be examined.

This task could be accomplished using the UNIX grep[8] tool to search for imported classes from the `reporting` package. However, it would have been more difficult to accomplish:

- It would be difficult to ensure all calls to the `reporting` package were captured without introducing false positives, such as a comment containing the word "reporting." This is particularly true in object-oriented systems as grep cannot compensate for polymorphism.
- The number of calls made from one class to another would be difficult to ascertain, so it would be difficult to prioritize which classes to focus on first
- It would be difficult to see whether there are any patterns in the calls. For example, if calls were being made from a class and its subclasses to another class, this might not be apparent.

3.2 Refactoring a Class

Prawn can also be useful when refactoring a class or set of classes. Refactoring may be necessary to introduce additional functionality, to improve the performance of some aspect of the system, or to change the design so that it is more extensible.

In the CSSIS system, one of the key modules is a `user` package that contains the functionality for different kinds of users of the system. Changing any of the classes in the `user` package would affect many classes in the rest of the system. Expanding the nodes until the children of the `cssis` package are shown, this becomes clear; four packages call the `user` package, including the `reporting` package and especially the `advising` package.

[8] `http://www.gnu.org/software/grep/grep.html`

Given the large number of calls between the `advising` package and the `user` package, more investigation would be needed. This would especially be the case since the part of the advising module uses Java Servlet pages, and other part is part of a Java application. Both parts rely on a layer of the module that reads and writes information to a database. Expanding the `advising` package, the developer can see that the distribution of calls from each of these parts to the `user` package is roughly equal. Hence, the developer will need to spend time with each of the developers responsible for these parts to determine how the effort of the changes that are planned.

With Prawn it is possible to quickly ascertain the difficulty of making a particular change to a system. Estimating the difficulty of making a change and the time required to do is not a trivial task. Even with domain knowledge of a system, having an interactive tool like Prawn allows a developer to make predictions with greater confidence, since their analysis is grounded in facts taken from the implementation.

3.3 Comprehension and Extension

While the previous scenarios have relied on expert knowledge of a system, Prawn can also be used by developers to learn about the structure of a system and to gain insight into how the system could be changed.

JHotDraw is a well-known drawing framework created by Eric Gamma as a demonstration of how a well designed system can take advantage of design patterns. It consists of 90+ classes and over 20 000 lines of code. One of the authors had basic knowledge of JHotDraw, but had never examined the source code. Using Prawn, he attempted to determine what changes would be required to add a Koala figure to JHotDraw's palette of figures.

After expanding the `CH.ifa.draw` node (see Figure 6), the author noted the presence of a `figures` package and expanded it (see Figure 7). To reduce the number of calls being displayed, he first hid all the calls at the `CH.ifa.draw` level, then hid calls within the `figures` package as well. He then proceeded to highlight the structural calls within the `figures` package (see Figure 8).

Next, the author noted that `KoalaFigure` would extend the `AttributeFigure` class. He then decided to see what classes within the `figures` package called `AttributeFigure` and its subclasses. He accomplished this by moving the pointer over each of these nodes and highlighting all their calls. After this, the author decided to group the non-`Figure` classes according to whether or not they interacted with the `Figure` hierarchy (see Figure 9). It was useful for the author to zoom out so that more information could be displayed on the screen.

Based on this information, the author identified several classes that call each `Figure` and thus would have to be modified for the addition of the new figure. He also observed that:

- What would seem to be related classes did not call the `Figure` classes equally. For example, there were several classes ending in "Handle," but only some of them made calls.

- There were several classes ending in “Connector,” but there was no superclass evident within the package. Since these classes all had references to `CH.ifa.draw.standard`, one could surmise that their superclass would be located there.

From here, the author could have continued to determine what packages are called by the `Figure` hierarchy, and what packages call the it. Although the author would not know exactly what changes would have to be made to add the figure, Prawn provided an indication of where the changes would likely be made.

4. RELATED WORK

Visualization has been used as a method to help understand [9], debug [1], and maintain [2] complex software systems. Visualization tools may show dynamic (runtime) behavior or static (structural) data. Further information about the software change process (from information stored in CVS or bug-tracking systems) can be visualized in order to determine related aspects of the system [2]. Both high and low-level aspects of the system can also be visualized.

Low-level visualizers try to examine every call that the system makes and are often used to help locate performance problems [11]. High-level visualizers often try to extract a system's architecture and coarse structure to allow developers to determine whether or not design contracts have been broken. Visualization systems can also help to identify the presence, or absence, of system properties such as reusability and maintainability [9]. Graph-based visualizations can be useful for identifying dependencies between code artifacts [12] which can help developers to quickly identify pieces of code which depend on one and other. This is particularly important in polymorphic object-oriented environments.

Prawn takes a static, offline approach which limits it from understanding how the program actually behaves at runtime. As such, tools such as AVID[9] and Jinsight[10] are better suited for identifying performance bottlenecks and system hot spots. Prawn also is not designed to be used as a visual debugger. However, since dynamic information is not used, the code does not need to be instrumented or otherwise altered to collect trace data. Prawn only requires access to a JAR file containing a compiled representation of the system in question in order to visualize it.

Sugibib[11] optimally draws UML diagrams from source code. While Prawn would have liked to use this layout package as a library for node placement, the source of this system is not in the public domain. Sugibib modifies the Sugiyama [8] layout model and augments it with changes that adapt it to be optimal for software systems. This tool also demonstrates [4] that general approaches can be made more effective when they are tailored specifically for software visualization.

The Seesoft [2] system employs a different visualization technique to show how a system has evolved over time. Seesoft shows individual lines of code as pixels on the screen. Links between classes (files) are not shown in the system. This allows Seesoft to scale to systems greater than 1MLOC. The scalability of this system is its greatest strength, although details about the complex interactions in the system are lost by this approach.

[9]`http://www.cs.ubc.ca/labs/spl/projects/avid.html`
[10]`http://www.alphaworks.ibm.com/tech/jinsight`
[11]`http://www.sugibib.de/english.html`

4.1 SHriMP

The SHriMP[12] tool from the University of Victoria was the inspiration for Prawn. Our intention was not simply to reimplement SHriMP, but to find ways to improve on it.

SHriMP allows developers to navigate their source code via a graph representation within a fish-eye view. It is an offline viewer that allows the system to be navigated via the linkages between classes. Users also have the ability to link directly from the graph representation to the source code. The primary display mechanism of SHriMP is a nested graph. This approach lends itself particularly well to visualizing software, as often elements are nested inside one other.

SHriMP provides three layout algorithms, two of which are provided by Prawn. However, the SHriMP layout system can employ only one layout at a time. Prawn provides radial layouts, tree layouts, and grid layouts at times when each seems most appropriate to best present the nodes being drawn, as well as make the most effective use of available screen space.

The fisheye view employed by the SHriMP project is more developed than the Prawn zooming model. Both Prawn and SHriMP allow a developer to select and enlarge any node while maintaining the context in which it is placed. However, SHriMP accomplishes this without having to resize the ancestor nodes; the siblings of the node are resized to accommodate its new size. In Prawn, ancestors are resized if the new size of the node is sufficiently large. As a result, nodes are moved off-screen as the graph is navigated to deeper levels. Neither tool allows a developer to arbitrarily resize a node, which would greatly increase usability.

Both SHriMP and Prawn have arc filtering capabilities, and both use colour to distinguish between different kinds of arcs. However, Prawn gives the developer more flexibility in showing and displaying arcs, with its Arc Threshold setting, Arc Width setting, and ability to arbitrarily show, highlight, or hide arcs from a node. In SHriMP, arc filtering is applied to the entire graph.

In practise, the authors found SHriMP difficult to use. Viewing the CSSIS project required the source code to be parsed using third-party tools, with the results then converted into the Rigi Standard Format (RSF) that is used by SHriMP. Loading the CSSIS RSF in SHriMP took approximately 3 times longer than loading a JAR of the CSSIS class files in Prawn, and used 4 times as much memory. The authors also were unable to get SHriMP to display CSSIS in a structured manner; the view presented showed all packages, classes, interfaces and methods in a single graph, rather in the hierarchy the authors expected.

5. EVALUATION

[12]`http://shrimp.cs.uvic.ca`

The current implementation of Prawn has many strengths and weaknesses. Several of these weaknesses are not due to technical limitations, but time limitations that prevented us from implementing them. Work items that are more substantial are discussed in the future work section of the paper. The evaluation section takes the form of a task-based analysis and discussion where Prawn's suitability for each task will be discussed.

We have used our tool throughout its development and during our scenario generation and we have determined that it successfully scales to visualize systems of jEdit[13] size with reasonable response times. jEdit is an open-source text editor written in Java. It is composed of 20 packages containing 644 classes, with approximately 23 270 method calls between the classes. It should also be noted that Prawn was also able to load and visualize the Java Runtime Library, which contains 314 packages and 8216 classes, although response times on an Athlon 750 were on the order of minutes for expanding a node. Nevertheless, we were able to use Prawn to accomplish a number of tasks which would have been cumbersome with traditional techniques, and would not have been possible with other similar visualization packages due to the scaling issue.

5.1 Prawn Strengths

We believe that Prawn has three core strengths: its ease of creating a visualization, its ability to facilitate the exploration of a large system, and its ability to reduce the complexity of a visualization of a system.

These strengths are contingent on providing the most relevant information possible to the developer in a way that is easy to understand and manipulate. The focus+context technique used by the tool allows developers to focus on the nodes they are most interested in. The three layout algorithms help to optimally place nodes so that determining which nodes to investigate next can be inferred as easily as possible. The Tree Layout in particular performs a clustering operation by grouping nodes based on the inheritance relationships between them. These layouts are augmented by the colour and transparency elements of the nodes which allow for easy delineation between the different types of artifacts on the screen and the relationships between them. The interaction model allows for simple navigation of the nodes, as well nesting levels within nodes.

5.1.1 Creating an Initial Visualization

The overhead required to start Prawn and navigate the source code is minimal. All that needs to be done is for a JAR of the program's byte code to be provided on the command line. Immediately, the Prawn canvas loads along with a simple command reference and control panel. With this low overhead, systems can be quickly visualized at any stage of development to provide updated information about the system. For example, the authors used Prawn to visualize itself during development and typically it took less than 30 seconds to create the JAR file, start the visualization program, and start navigating on an Athlon 750.

5.1.2 Large System Exploration

Exploring large systems requires that developers can have fine-grain control of the amount of information they are seeing, as well as the ability to focus on the aspects of the system they are interested in, while ignoring the rest. Prawn facilitates both of these goals through its arc elision features and its focus+context layout approach.

In the Comprehension and Extension scenario, the author was first able to elide all arcs except for the structural arcs within the `figure` package to gain insight into the key nodes in the package. While the author examined the `figure` package in detail, other packages were not shown in detail so that they were not a distraction. Later on, the author could have expanded packages as necessary to understand how classes in the `figure` package interacted with them.

Another example of this is the Java Runtime Library. This large collection of classes is certainly cumbersome to explore by traditional techniques. However, if a developer were to take an interest in only the `java.util` package they could focus only on that package and ignore the majority of the classes in the runtime. Also, they could focus only on how the nodes were placed to see how they are related to one another. In the case of `java.util` Prawn clusters the nodes into 6 primary groups (Collection, SortedSet, Iterator, Map, Map$Entry, and RandomAccess). This clustering can be seen in figure 10. From this grouping, it can be seen that the Collection, Iterator, and Map clusters are the largest, and as such could be considered the most important, nodes with which to begin any code examination task.

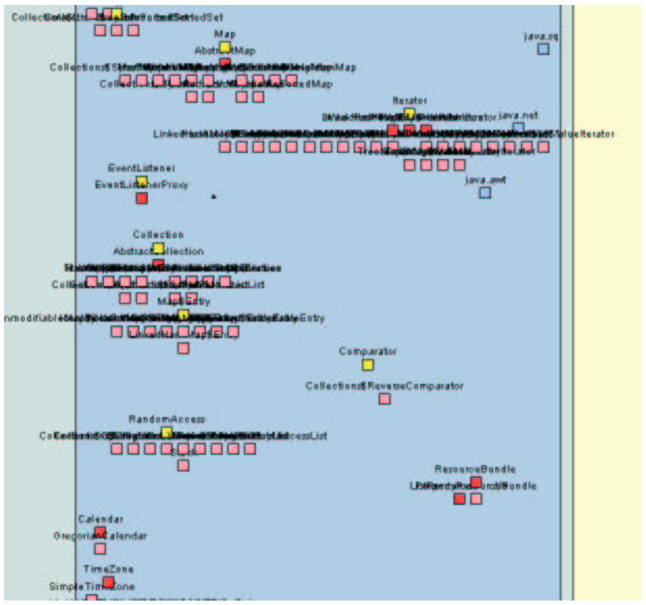


Figure 10: Part of the `java.util` package

While experienced developers usually do not need to explore a system in general, often novice developers require extra support, especially when learning how to use complex frameworks [3] [5]. This is because frameworks often lack adequate documentation to help new developers learn how to use them. However, when users are developing code within a framework, their simple systems are suddenly a

[13]`http://www.jedit.org`

small part of a large one which is much harder to understand. Using Prawn, these novice developers can see their system in full detail, while showing only major interconnections to the framework, which can allow the developer to see how their work fits into the overall structure of the system. This can help the developer to understand the implications of code they are writing, as well as quickly look at more detail inside the framework itself if greater detail is needed.

5.1.3 *Reducing Complexity*

One of the primary problems with graph representations is that they quickly become cluttered and do not scale well [9]. Prawn separates the task of reducing the complexity of the nodes and the complexity of the arcs into two different problems. The layout algorithms take care of the placement of nodes. This includes clustering by inheritance hierarchies, and using different layouts based upon the kind of nodes that are being visualized. However, the authors found that the volume of arcs were the most problematic for large systems. Therefore, three simple techniques were introduced to make the graph easier to understand. The arc threshold feature allows arcs of less than a certain weight to be hidden. This is particularly effective when a developer is trying to identify the most important classes in the system, without clutter. The arc widths feature can be used when it is important to preserve all available information while highlighting only the most important arcs. The third technique allows developers to hide specific classes of arcs. These classes are: `All Arcs`, `Incoming Arcs`, `Outgoing Arcs`, and `Structural Arcs`. These arc elision features can be chosen for whole packages or just individual nodes depending on what the cursor is over when the key is selected. Further, the mouse can be hovered over any node in order to see the full information. This allows a clean high-level view while allowing the low-level details to be quickly determined.

5.2 Prawn Weaknesses

The primary weaknesses of Prawn are directly related to some of its primary strengths as well. As the authors developed the tool they implemented the elision and layout mechanisms that exist in the tool. However, several extensions are required to make these metaphors fully functional, and to maximize their effectiveness.

The focus+context model could be dramatically improved to place the context nodes in a more optimal manner. The current algorithm pushes them to the periphery and does not consider any form of rank on these context nodes. The clustering model could be aggregated to arcs such that arcs going to subclasses could instead go to a superclass or abstract class. This could help reduce the arcs going to subclasses, as well as reinforce the connections between different class hierarchies. The visual encoding scheme could be extended to include multiple glyph types which would help to greater differentiate the different nodes as well as supply extra information about them (perhaps connectedness, etc). Animated transitions could be used to resize the nodes when nested structures are opened such that the new nodes all fit on the screen at once. Also, the text labels on the nodes should be staggered in some way so that they do not occlude one another.

5.2.1 *Layout*

The layout, in particular, is a project unto itself. Simple layout strategies were identified and implemented in the project, but a great deal more work could be done in this case. Prawn does not adequately layout any nodes that are not involved in inheritance hierarchies. Therefore, simple utility classes are often grouped into a box layout at the top of a package structure. As these classes tend to be used throughout the code these box structures are often difficult to comprehend. Some form of weighted clustering / layout should be used so classes which are used often are given extra space so that it can be clear that they are important. Conversely, nodes which are tightly coupled by lots of calls between them could be clustered together to gain more of a function-based layout. Implementing advanced algorithms, such as those used by Sugibib could also increase the effective use of space by the tool. Also, our algorithm which keeps nodes from overlapping one another is a simple matter of collision avoidance, and doesn't try to optimally place these nodes. Prawn also makes no effort to reduce edge crossings which can unnecessarily complicate diagrams. Further arc elision by identifying arcs which are not part of structural or calling relationships and hiding them could also help to reduce the number of edges on the graph.

5.2.2 *Artifact Manipulation*

When navigating the system, often groups of nodes need to be examined at once. Currently, nodes can only be selected individually or by packages. Allowing custom node groups to be selected and manipulated as one would certainly allow for custom layouts to be made more easily. Also, arcs are not updated when nodes are in transition, only when they are moved to new locations. This can make placing a node optimally a trial and error operation. However, it is unlikely that all arcs could be dynamically updated as the node is dragged. A better solution would be to dynamically update as many arcs as possible, giving highest priority to structural arcs and those with the most calls or those that are highlighted. This general approach was used in the H3 system [10]. Supporting text searches, and classification searches (selecting all interfaces for example) could also help to select subsets of nodes for group manipulation.

5.3 Lessons Learned

Over the course of this project, we learned several things:

- Creating an effective layout algorithm for placing nodes in a graph is a very difficult problem.
- Choosing an effective colour scheme is difficult, once the needs of people who are colour-blind are considered. Even once the colours have been set, it is easy to overuse colour, resulting in a display that is highly cluttered. This was apparent in drawing arcs; arcs were originally drawn in very dark colours, which made it very difficult to read the visualization.
- Making the visualization scale beyond trivial examples required us to investigate how we could convey more information using a few properties, and how we could hide unimportant information.
- No matter how advanced the information visualization techniques are, if the tool requires a significant amount of effort to use, it will not be adopted in the real world.

- Zoom is not the perfect solution for everything. Zooming out allows more information to be displayed, but at the cost of increased clutter.

- Tackling a problem that has seen 10 years of research in the Rigi and SHriMP projects in the space of one month was slightly ambitious. But we did better. So there[14].

6. FUTURE WORK

Several work items remain which, if implemented, would make Prawn a much more complete and effective tool. The most important, and fundamental of these is integration into an IDE. This IDE integration would allow for even less overhead in visualizing systems. The integration would be a two-way process. Developers browsing the source could pull up a pictorial representation of the part of the system they were currently looking at, as well as go from the nodes in the graph drilling down into the source code. Advanced querying can also help to reduce the amount of navigation that is required in the system (for instance, find all occurrences of abstract classes which implement an interface and are extended by these three sample applications).

Further improvements in the visualization could be realized by only looking at subsets of the graph. Currently nodes can be collapsed but are still present. Visualizing only one or two levels from any single node could help to make the graph size smaller and limit the scope of navigation to nodes known to be somehow related to the concern being currently analyzed. Navigation could be simplified by providing an overview radar view, as well as making the links active (clicking on a directed link will take you to the target of that link).

Data flow information can also be integrated into the tool to see how data is accessed throughout the system. Currently method data is maintained by the tool but is not used for anything except for calculating dependencies between nodes. Further development could allow for drilling into specific methods in classes to focus only on the dependencies of specific methods, instead of whole classes. This could also be useful for determining fine-grain calls and called-by relationships.

Prawn could also annotate the visualization with additional information from CVS or a bug-tracking database, in addition to providing links into the source code from the visualization. Tools such as Hipikat[15] exist that actively provide this information to a developer, although they have not been used in this particular context.

7. CONCLUSION

We have developed the Prawn program visualization tool. This tool is a static, offline visualization system which is targeted at developers who need to get a better overview of how their system's components interact with one another. By leveraging the graphical node-link metaphor the details about how individual methods interact can be aggregated to classes, packages, and interfaces to reduce the cognitive load on the developer. The front-end was designed through an iterative process which took advantage of many information visualization techniques to reduce this cognitive load and increase the effectiveness of the tool. Through a simple set of scenarios we illustrated three simple tasks that Prawn can be used for during the development life-cycle. We believe that with some future work, this tool can be made into a low-overhead way to help developers quickly understand complex software systems.

[14]We are being facetious, of course

[15]`http://www.cs.ubc.ca/labs/spl/projects/hipikat.html`

8. REFERENCES

[1] Ronald Baecker, Chris DiGiano, and Aaron Marcus. Software visualization for debugging. *CACM*, 40(4):44–54, 1997.

[2] Thomas Ball and Stephen G. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996.

[3] G. Butler and P. D'enomm'ee. Documenting frameworks to assist application developers, 1997.

[4] H. Eichelberger and J. Wolff von Gudenberg. On the visualization of java programs. In *LNCS 2269, S. Diehl (ed): Software Visualization International Seminar*, pages 295–306. Springer, 2001.

[5] Gerhard Fischer, Scott Henninger, and David F. Redmiles. Cognitive tools for locating and comprehending software objects for reuse. In *ICSE*, pages 318–328, 1991.

[6] George W. Furnas and Benjamin B. Bederson. Space-scale diagrams: understanding multiscale interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 234–241. ACM Press/Addison-Wesley Publishing Co., 1995.

[7] S. Tilley H. Müeller, M. Orgun and J. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, 1993.

[8] S. Tagawa K. Sugiyama and M. Toda. Methods for visual understanding of hierarchical systems. In *IEEE Trans. Syst. Man*, volume 11, pages 109–125, 1981.

[9] Michael F. Kleyn and Paul C. Gingrich. Graphtrace–understanding object-oriented systems using concurrently animated views. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 191–205. ACM Press, 1988.

[10] Tamara Munzner. H3: Laying out large directed graphs in 3d hyperbolic space". In *Proceedings of the 1997 IEEE Symposium on Information Visualization*, pages 2–10. IEEE Computer Society Press, 1997.

[11] Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 326–337. ACM Press, 1993.

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

[13] J. Wu and M.-A.D. Storey. A multi-perspective software visualization environment. In *Proc. of CASCON'2000*, pages 41–50, 2000.

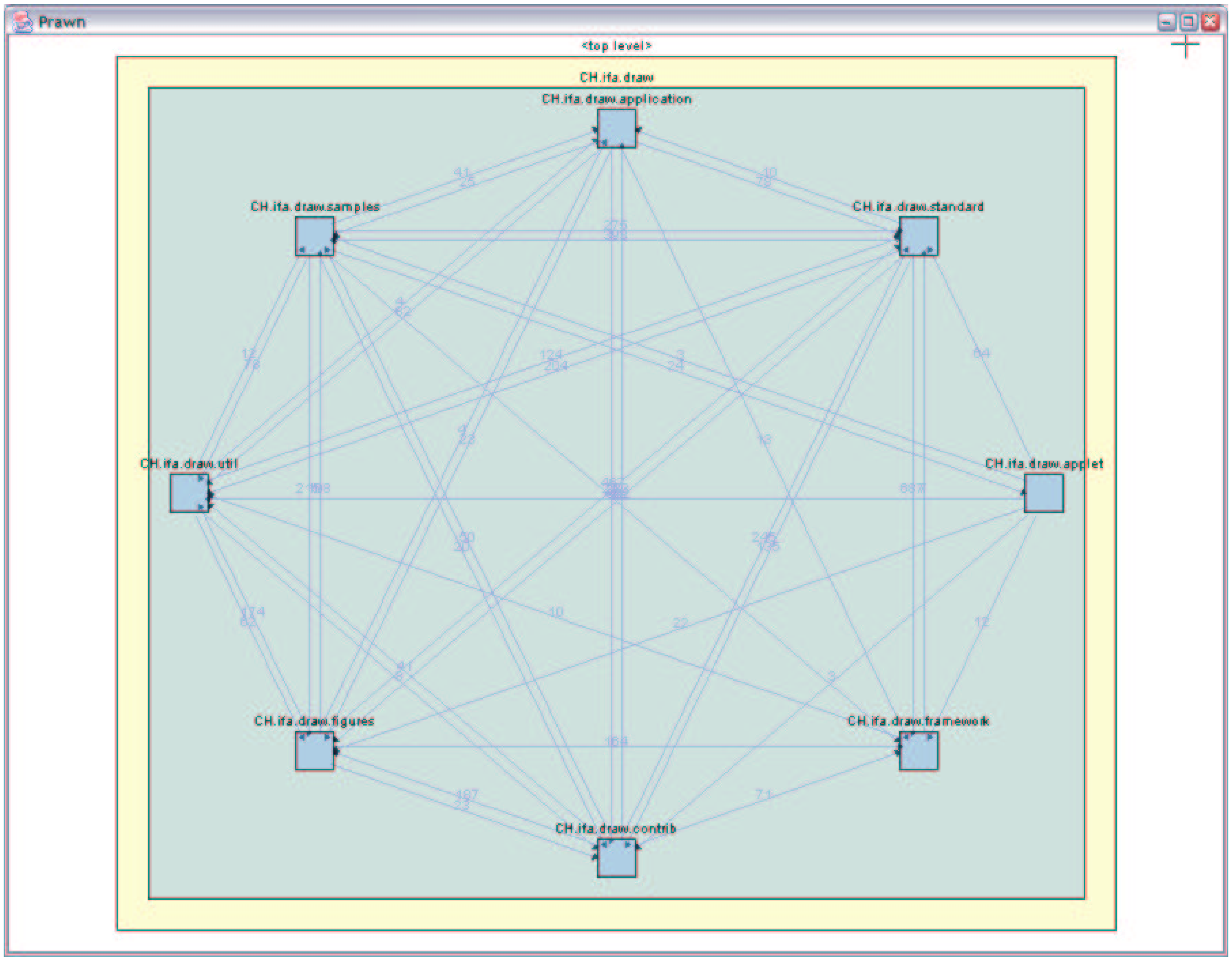


Figure 6: Prawn Canvas

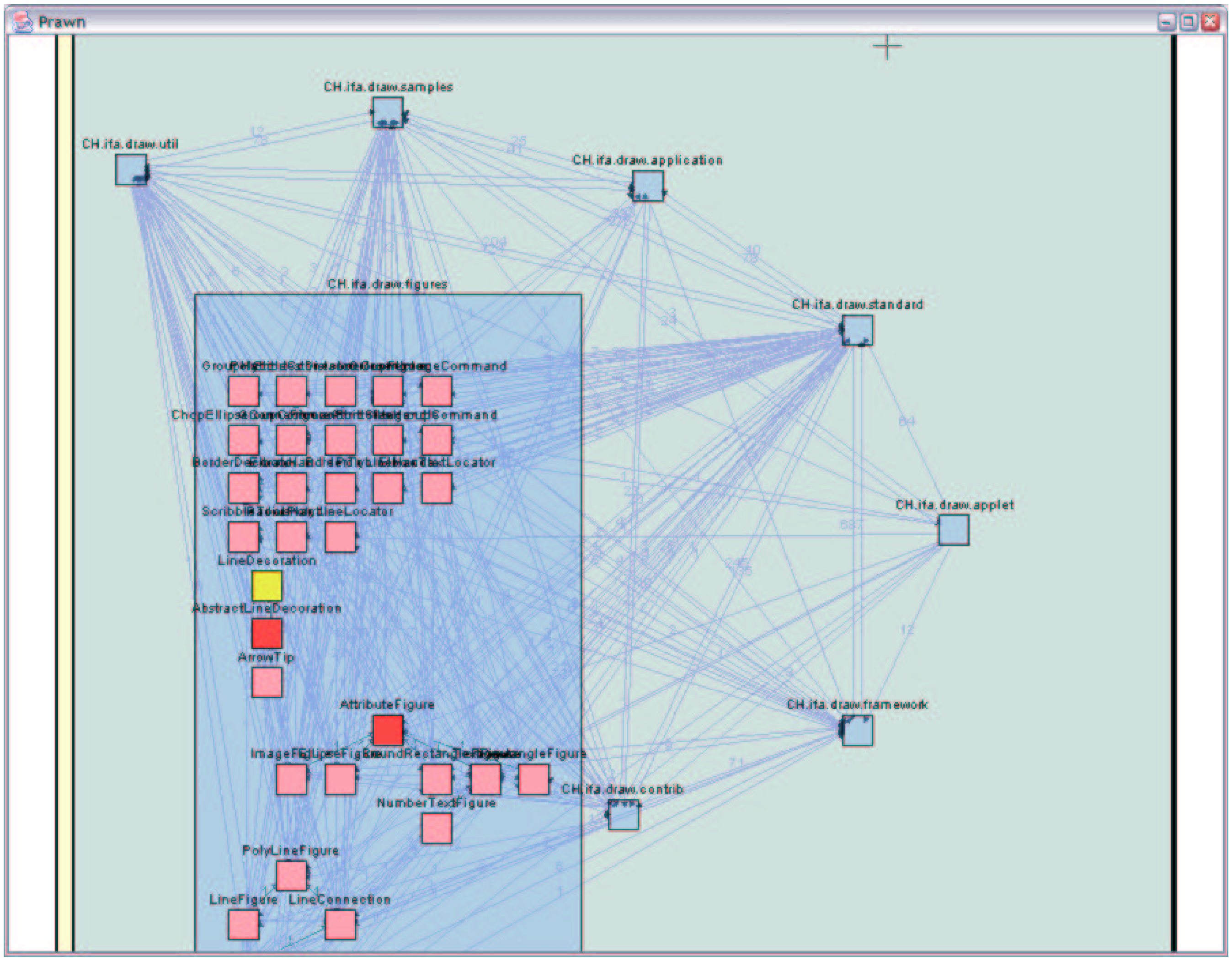


Figure 7: Prawn Canvas

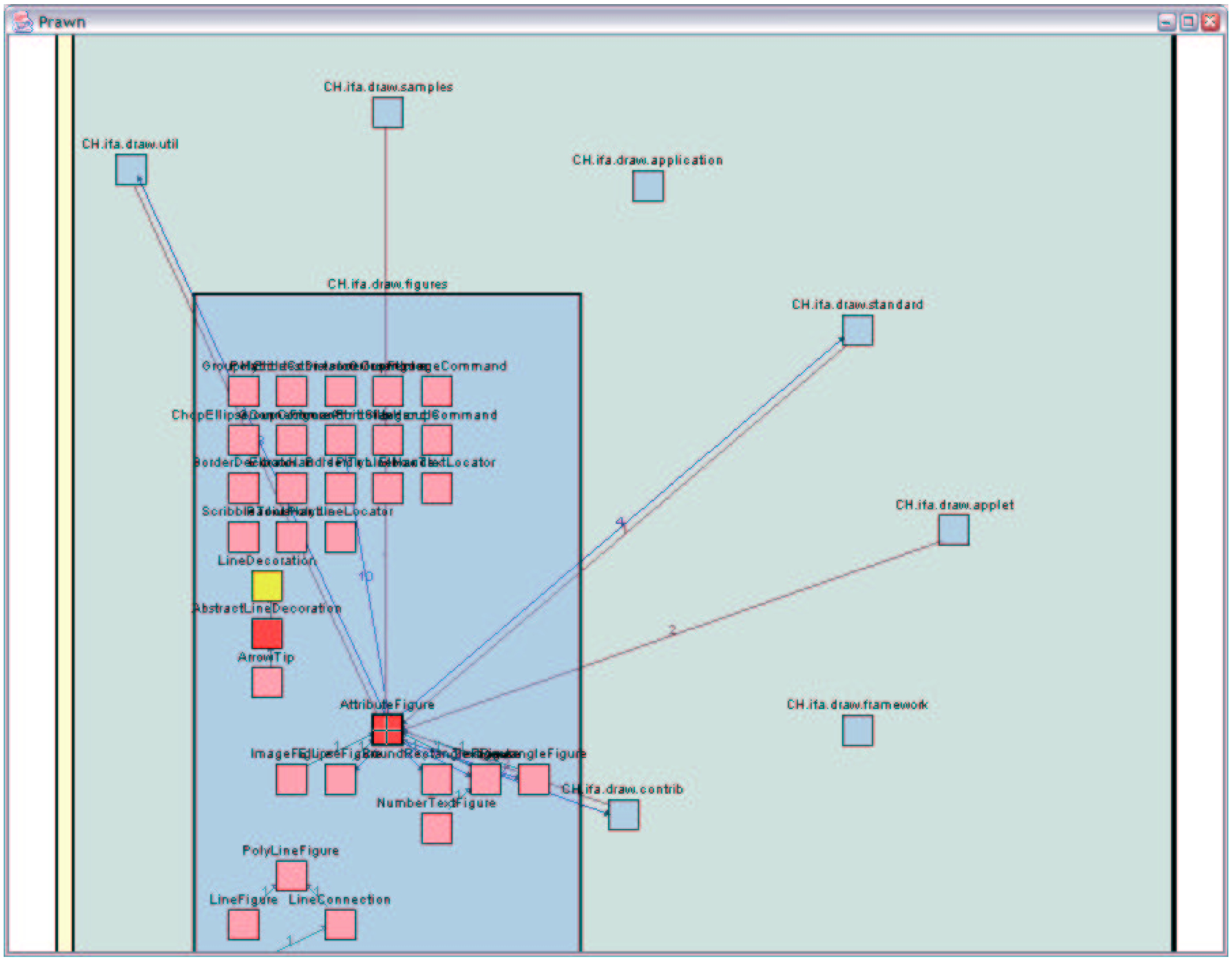


Figure 8: Prawn Canvas

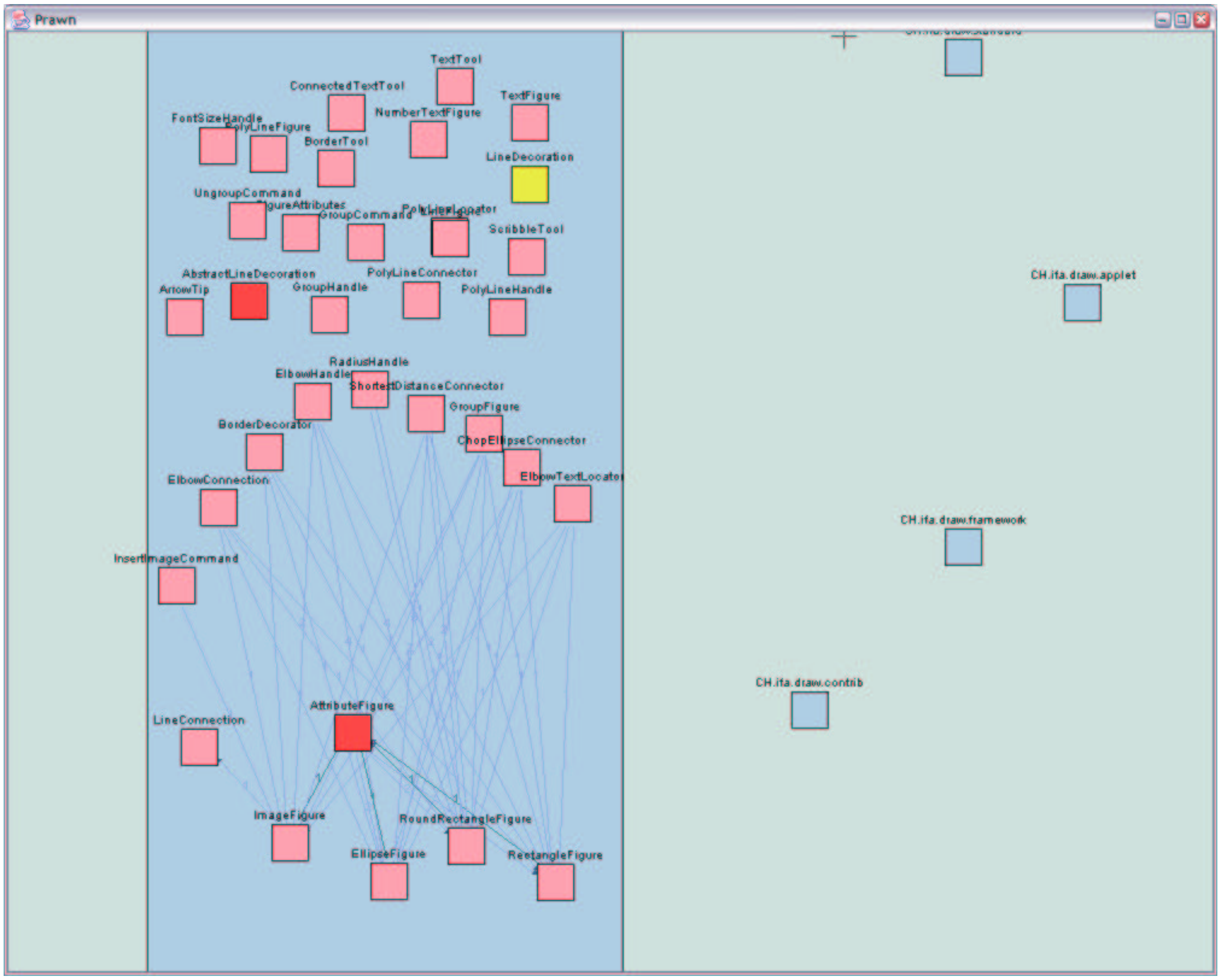


Figure 9: Prawn Canvas