MASSACHVSETTS INSTITVTE OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPVTER SCIENCE

Proposal for Thesis Research in Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy

Title: Dr. Jones: A Software Archaeologist's Magic Lens

Submitted by: Mark A. Foltz
MIT Artificial Intelligence Lab
200 Technology Square, Room NE43-808
Cambridge, MA 02139 USA

(Signature of author)

Date of submission: May 22, 2001

Expected Date of Completion: May, 2002

Laboratory where thesis will be done: MIT Artificial Intelligence Laboratory

Brief Statement of the Problem

Program comprehension remains a major bottleneck for software maintenance. When a programmer must understand a large legacy program whose documentation is scarce, out-of-date, or irrelevant, she must browse its source code to build an effective mental model of its structure and behavior. But source code is far from an ideal representation for this task.

I propose a system, DR. JONES, to help a programmer understand an unfamiliar program. DR. JONES is a magic lens over the program that reveals information that usually requires effort to extract from source code. This information is broken into tightly coupled views of the program's structure and behavior that, taken together, help the programmer build a good mental model of the program. DR. JONES presents these views as diagrams, to engage the programmer's perception in extracting knowledge about the program.

The design of DR. JONES will be motivated by user studies to find the kinds of knowledge programmers seek to extract from unfamiliar programs, and the strategies they employ to do so. DR. JONES will make use of techniques like multiscale browsing, design pattern recognition, and "software jiggling" to permit the orderly and conceptual exploration of an unfamiliar program.

DR. JONES: A Software Archaeologist's Magic Lens

Doctoral Thesis Proposal

Mark A. Foltz
MIT Artificial Intelligence Lab
mfoltz@ai.mit.edu
May 24, 2001

Abstract

Program comprehension remains a major bottleneck for software maintenance. When a programmer must understand a large legacy program whose documentation is scarce, out-of-date, or irrelevant, she must browse its source code to build an effective mental model of its structure and behavior. But source code is far from an ideal representation for this task.

I propose a system, DR. JONES, to help a programmer understand an unfamiliar program. DR. JONES is a magic lens over the program that reveals information that usually requires effort to extract from source code. This information is broken into tightly coupled views of the program's structure and behavior that, taken together, help the programmer build a good mental model of the program. DR. JONES presents these views as diagrams, to engage the programmer's perception in extracting knowledge about the program.

The design of DR. JONES will be motivated by user studies to find the kinds of knowledge programmers seek to extract from unfamiliar programs, and the strategies they employ to do so. DR. JONES will make use of techniques like multiscale browsing, design pattern recognition, and "software jiggling" to permit the orderly and conceptual exploration of an unfamiliar program.

Let us change our traditional attitude to the construction of programs. Instead of imagining that our main task is to instruct a computer *what to do, let us concentrate rather on explaining to* human beings *what we want a computer to do.*

Donald Knuth[1]

I'm like a bad penny, I always turn up.

Dr. Henry "Indiana" Jones

1 Sarah Singleton's Expedition

Consider the situation of Sarah Singleton, a programmer who has just joined the software team for a hockey equipment retailer. Sarah is asked to add wish list functionality to the retailer's Web site as soon as possible (of course).

First, she must understand enough about the site's software to ensure that her new wish list feature will interact properly with the existing code. So, she browses the code and documentation, and discusses the system with the programmers currently maintaining it. During the course of this investigation she builds a mental model of its workings: a "mental picture" of the structure (classes and APIs) and behavior (control and data flow) of the system.

A similar challenge faces any programmer who must understand existing software before she is able to maintain or enhance it. If she is lucky there will be well-written documentation, both within the source code and in external documents. But if documentation is incomplete, out-of-date, or irrelevant to the task at hand, then work remains to discover the program's structure and behavior and build a useful mental model. Knowing where to begin and what is relevant is half the challenge, especially in programs with tens of thousands of lines of code (LoC).

I propose research to develop a tool, DR. JONES, that assists a programmer in program comprehension. A guiding metaphor is that the programmer is a software archaeologist who must investigate a mysterious device from long ago to gain insight into its structure and behavior. DR. JONES acts as the archaeologist's magic lens, allowing her to immediately see relationships and patterns in the program that would ordinarily require much more effort to deduce.

DR. JONES analyzes a program[2] and presents interactive, graphical views of it to the programmer. These views are designed to make explicit the kinds of concepts and patterns that programmers use to reason about and describe their programs. The programmer explores the program through DR. JONES' magic lens to incrementally build an effective mental model of it.

1.1 The Thesis

The thesis of this research is that a programmer's mental model of a program is organized around a small number of hierarchical, tightly coupled views of the program's structure and behavior, the

[1]In *Literate Programming*, (Knuth, 1991).

[2]Here a program is assumed to be an object-oriented program written in Java (Joy *et al.*, 1998) or a similar language.

design patterns used in its development, and concepts from the application domain.

A program's structure is the information contained in a static snapshot of the program, which changes little during execution. A program's behavior captures the dynamic aspects of its execution, such as its objects' lifecycles and threads' timelines. Design patterns are recurring programming motifs that usually involve both structure and behavior; for example, iteration over a list, or an object factory. Concepts from the application domain relate the program's structure and behavior to the functionality it is intended to implement.

This research has two goals. The first goal is to determine which views can be effective for program comprehension through a series of exploratory user studies. The second goal is to develop Dr. Jones, a system which lets a programmer explore these views of an unfamiliar Java program.

The principles guiding my approach to this problem are:

- To attend to the concepts and strategies programmers use to understand programs, to characterize the gap between the program's text and its mental model;
- To employ visualization as a means of presenting program views, to engage the programmer's perceptual faculties for interpreting complex information;
- And, to permit interactive navigation and incremental understanding of the program, so the programmer is not overwhelmed all at once with the its complexity.

1.2 Proposed Contributions

Much prior research on program comprehension tools has been undertaken. Many approaches are based on design pattern recognition in programs (Rich and Waters, 1989), as well as structural or behavioral visualization (Pauw *et al.*, 1998). Others have also taken a cognitively motivated approach to the development of comprehension tools (Storey *et al.*, 1997).

The main contribution of this research is an approach that closely links *program comprehension* to *program visualization*. I seek to examine how programmers think about programs, and use this knowledge to present more effective program views through Dr. Jones' magic lens. The goal is to better bridge the gap between the source code facing the programmer and the the mental model she needs to maintain and enhance it.

The contributions will potentially impact three established research areas:

1.2.1 Artificial Intelligence

A long-term goal of Artificial Intelligence is the discovery of methods and representations that allow computers to reason about complex devices and systems. Dr. Jones' program views will be enhanced by the automated recognition of meaningful patterns in the program. This will require the development of novel methods and representations for reasoning about software systems. And, since programs can simulate the behavior of many kinds of systems, it may provide insights about reverse engineering in a variety of domains.

1.2.2 Software Engineering

This research will also examine the concepts and strategies that programmers use to comprehend unfamiliar programs. These findings can inform the development of other software engineering tools, design methodologies, and the education of software engineers. For example, a CASE tool that allows programmers to express their design in terms of familiar concepts should feel more natural to use and act as a more competent design assistant. DR. JONES contributes to a growing research toolset for software maintenance, and assessing its successes and shortcomings will assist future research in this area.

1.2.3 Design Rationale Capture and Use

Software maintenance activity is a primary motivation for the capture of software design rationale. The program views provided by DR. JONES will augment manually generated rationale captured during the design process, such as requirements, specifications, and discussion of alternatives. Although DR. JONES' primary purpose is not to reveal the complete design intent of a program, it could help the programmer infer some aspects of design rationale that are missing. (Others have shown that it is possible to infer some design rationale by examining the behavior of mechanical devices (Raghavan and Stahovich, 1998)). This research looks at the design process from the artifact backwards, thus suggesting which kinds of rationale are most useful for program understanding, and should be captured upstream in the design process.

1.3 Back to Sarah Singleton

Let's return to our software archaeologist and her wish list requirement, now supposing she had DR. JONES. In Figure 1, she is browsing a view that traces the timeline of a thread handling a Web request. Her initial investigation reveals that these threads are handled by subclasses of `Transaction` in this system, according to the kind of request. She decides she must add two new subclasses of `Transaction`, `AddToWishList` and `RemoveFromWishList`. She browses the control flow in and out of existing subclasses of `Transaction`, to understand how they interact with other parts of the system. With this knowledge, she can begin to design the APIs for the new classes.

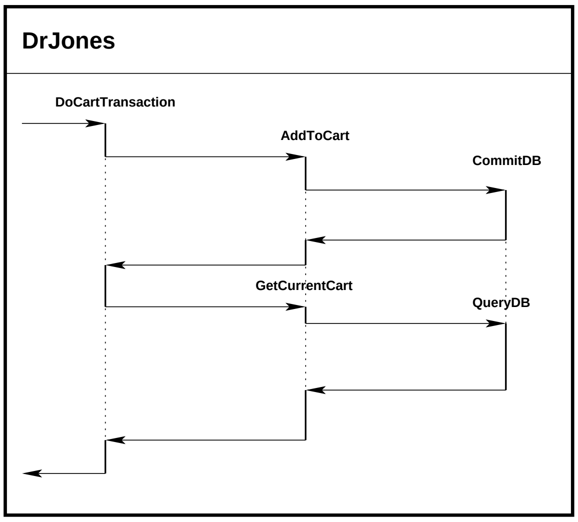

Figure 1: A mockup of DR. JONES showing an execution thread handling a Web request in a hypothetical Web site.

1.4 Outline

The remainder of this proposal discusses:

- The difficulties encountered in program comprehension when browsing source code;
- User studies to find the strategies and concepts programmers use to comprehend programs;
- A sketch of the user interface for DR. JONES;
- Criteria of success for DR. JONES;
- Possible surprises that may alter the course of this research, its future prospects, and related work;
- And a schedule of research (in an appendix).

2 Source Browsing Breakdowns

An approximate synonym for program comprehension is reverse engineering: an attempt to infer a program's design by studying the resulting artifact. This task is made difficult because the source code is the result of design, but makes explicit little about the design itself. Experienced programmers bring to bear an immense amount of knowledge to compensate, including the syntax and semantics of the language, design patterns and algorithms, the application domain, and the purpose of the program. Unfortunately, source code is a difficult representation to work with for this kind of reasoning.

2.1 The Trouble With Source Code

The main difficulty is that source code is an atomic and linear program specification, while programmers' mental models are holistic, made up of densely interrelated concepts. Despite claims that good program code is almost "self-documenting," experience shows that it has many drawbacks. In particular:

Code maximizes modularity. A good software design maximizes the modularity of the resulting program. In fact, this is the one of the underlying design philosophies of object-oriented programming: *break the problem down into the smallest pieces, and assemble a solution from them.* But this causes information about how the pieces are used to be distributed throughout the program. The programmer must then look in many places to reconstruct their interactions and get a picture of the overall behavior.

References are nonlinear, source code is linear. A major part of a program's power is its highly nonlinear and nonlocal referential structure. But this means it is usually not possible to browse an unfamiliar program's call graph without forward references to unseen classes. And, a single method may refer to many other classes, introducing a high branching factor to resolve all the references. Yet, source code affords only linear browsing, requiring the programmer to frequently search through other files and directories to locate class definitions. Hypertext links from class names to their definitions can automate this process (Kramer, 1999), but code navigation remains difficult and taxes the programmer's working memory (Zayour and Lethbridge, 2000).

Lack of context and overviews. A problem related to source code's linear nature is the small window onto the program offered by textual browsing. The programmer must retain the context of the visible program fragment in working memory, such as the enclosing method and class and the variables in scope. And, it is usually hard to get a "bird's eye view" of the program showing a structural overview. Integrated development environments help by presenting a browsable hierarchy of classes and methods, and other work has presents source code maps highlighting programmers' changes (Eick, 1998), but providing contexts and overviews suited particularly for comprehension remains a challenge.

Finding where to start and where to go is half the challenge. Finding the logical starting point(s) to begin reading source code is difficult, especially in high LoC programs. (The `main()` method is not always the best place.) A programmer who delves into a large program must choose which parts are relevant to her task and prune her browsing accordingly. For me, a typical program comprehension session begins with a period of flailing about and false starts, until I develop a clear enough picture of the program to browse in a more directed manner.

2.2 Overcoming Browsing Breakdowns

The difficulties encountered when reading a program's source code can be summarized by comparison with well-written documentation describing the same program. The documentation would first state the purpose of the program and present a structural overview of its classes and their relationships before delving into their implementation. If it mentions a class that has yet to be defined, a short explanation of its function would be provided. Use cases would illustrate how the classes interact to provide basic functionality. The documentation would present the program in a manner optimized for comprehension, not execution.

Obviously, reading good documentation is preferred to reading source code. Unfortunately, good documentation is scarce. DR. JONES is intended to capture some of the aspects of good documentation, like overviews and orderly exploration, in an interactive environment. Generating human-quality documentation of an undocumented program remains a long-term goal for this line of research.

Programmers adopt a variety of strategies for overcoming the limitations of source code browsing, to extract relevant and useful information about the program. The next section describes planned user studies to collect and analyze those strategies.

3 Comprehension Strategies and Mental Models

DR. JONES will provide program views to help the programmer build a good mental model of the program. But determining the content of those views – that is, what kinds of program knowledge constitute the mental model – is a difficult and uncertain task. It varies substantially across applications, organizations, and individuals.

We start with the assumption that mental models of programs are grounded in at least two kinds of knowledge. The first is the semantics of Java and the object-oriented paradigm, including concepts like *objects*, *threads*, and *exceptions*. The second is the semantics of the application domain, the part of the world the program is modeling. A good mental model allows a programmer to explain the functionality of the application in terms of the architecture of the program and the semantics of Java.

To understand these mental models more deeply, a place to start is anecdotes and introspection. When I recently needed to understand a large Java system for a Web site, given little useful documentation, I began by mapping its structure. I drew a diagram showing the inheritance and multiplicity relations among its classes, and included some of the classes' key methods.[3] Then, I mentally simulated the system by tracing the methods called to process and generate different Web pages. (This turned out to be difficult because every Web page in the site acted as an independent starting point for the program.)

In this case, mapping the structure of *is-a* and *has-a* relationships in the program was a step essential to my comprehension, implying a program view containing those relationships would be useful. But other programmers will use a variety of other strategies, suggesting many other ways of viewing a program.

3.1 Exploratory User Studies

To gain a broader view of comprehension strategies this research will conduct a series of exploratory user studies. The goal of the studies is to learn how different programmers go about software archeology, and the kinds of program knowledge they gather. The basic method will be to give a subject a small program and ask them to understand the program well enough to maintain it. They will be given a task to direct their exploration, such as redesigning the program to generalize an aspect of its functionality.

While they are browsing the program, we will encourage them to "think out loud" and explain the intent of their actions. The subject will be videotaped, and we will collect any marks made on paper or whiteboards. The subject will browse the source code and documentation as a granular hypertext to facilitate the capture of her browsing behavior.

After the browsing session, the subject may be asked to explain the program in her own words to the experimenters. Also, subject will be asked about the strategies they used to understand this particular program, and any general comprehension strategies they find especially helpful.

The purpose of the studies is not to measure performance or to produce a cognitive model of program comprehension in these conditions. Instead, the results will be analyzed to collect examples of comprehension strategies and the concepts and patterns the programmer recognizes in the program.

3.1.1 Preliminary Results

A pilot study with two subjects from the AI Lab has already been carried out. The task for this study was to design a small program given a problem statement, as opposed to examining an existing program. However, the subjects were later asked to explain their designs to the experimenters, and two interesting observations fell out:

1. Explanations of behavior often proceeded from the user interface backward, tracing the execution path in response to a user action (i.e., clicking a button).

2. Subjects used pseudocode to design and describe their program, attesting to the utility of representations that are similar to program text, at least during the early stages of design.

These results are preliminary, and many more insights like them are needed to make general statements about programmers' mental models.

[3]Unfortunately, the diagram is lost, as I would like to have included it here.

3.2 From Strategies to Models

The purpose of a comprehension strategy is to select some information from the program text, combine it with background knowledge of Java and the application, and improve the programmer's formative mental model. For example, the structure mapping strategy I employed selected the inheritance and multiplicity information from the source code of class definitions, and used it to augment my knowledge of the functional and multiplicity relationships among classes.

Each of the strategies identified in the user studies can be subject to the same analysis. The goal is to characterize the kind of information the programmer is pulling from the program:

- What parts of the source code did she use to derive the new knowledge?
- Is she using background knowledge of Java, the domain, or both?
- How difficult would it be to extract and present that information automatically? Will programmer input be needed?
- What is the best way to view that information? How is it coupled to other types of information the programmer seeks?

3.3 From Mental Models to Views

When a type of desired program information is identified in the user studies, it will motivate the inclusion of a program view in DR. JONES. The view will make visible and explicit the relationships and patterns sought by the programmer, and hide irrelevant information. If possible, the view will make use of graphical and spatial visualization to transform difficult, textual program comprehension tasks into a perceptual ones.

We expect there to be multiple views, and that they will be *tightly coupled*: the information in them will be densely interrelated, and the programmer will want to switch among them frequently. The design of DR. JONES must be sensitive to this fact.

3.4 Variations in Mental Models

Mental models of programs are needed to serve a variety of purposes for a variety of applications. Below we consider some of the ways mental models vary according to the software maintenance task and the application domain, and how DR. JONES plans to accommodate those variations.

3.4.1 Variations According to the Task

The knowledge a programmer needs is usually driven by a particular software maintenance task. The task determines which parts of the program she examines and what knowledge she derives from them.

Some tasks, like optimization and debugging, require the programmer to locate one small part of the program and make pinpoint changes. For these tasks global understanding of the program can help the programmer narrow down the portion that needs changing, but is not strictly necessary (Erdos and Sneed, 1998). Other tasks usually require global understanding, such as redesign, enhancement, refactoring and integrating subsystems, and documentation writing.

This research focuses on enhancing the programmer's global understanding, but at varying levels of detail according to her task. Tools like profilers, bug finders, and design assistants that assist particular software maintenance tasks could be integrated with DR. JONES to work within the views it provides. However, replicating their functionality will not be the focus of this work.

3.4.2 Variations According to the Application

Essential to the understanding of any program is knowledge of its domain - the part of the world that the program is modeling. Imagine trying to understand source code whose identifiers had been replaced with nonsense names, and furthermore, no information about the program's purpose was available. It would still be possible, but much more difficult, as reverse engineering the domain as well as the program would be necessary.

Domain knowledge helps understanding by creating expectations about the objects and actions one would find in the program. For example, if the programmer knows the program is a card game, she can infer the likely presence of objects like `Card`, `Deck`, and `Hand`, and methods like `deal()` and `shuffle()` even before looking at the source code. (Of course, it helps if the code writer has chosen intuitive, mnemonic names for these identifiers.)

A good object-oriented program design closely models the objects and relationships in its application domain. Our preliminary user studies and other research suggest that there are two kinds of program comprehension (Storey *et al.*, 1997). Bottom-up reasoning uses domain-independent knowledge of Java semantics and object-oriented programming, while top-down reasoning uses knowledge of the application domain's objects and actions to seek analogous structures in the program.

Thus, one significant part of a programmer's mental model is an analogy between the objects and actions in the domain and program structures. An ideal program comprehension tool would recognize the application domain and present the program in those terms, doing the work of analogy-forming for the programmer. Unfortunately, this means that a different kind of knowledge is needed for each kind of application. Customizing a generic tool with application-specific knowledge can create a second code base that must be maintained and debugged as the software evolves (Sayyad-Shirabad *et al.*, 1997). One possibility is dividing the work between the tool and the archaeologist (as dicussed below).

3.5 The User's Model Versus the Programmer's Model

The user's mental model of the program is distinct from but closely related to the programmer's mental model. In particular, the user is concerned with how his intentions map onto the affordances of the interface, and how the affordances activate the program's functionality. He is rarely concerned with how the functionality is implemented.

The programmer, on the other hand, seeks a mental model of the program that that can be used to account for all aspects of its behavior (including internal state) and predict the impact of changes to its source code. Because one goal of the programmer is to preserve the user's mental model of the interface, the programmer usually must be familiar with that model as well. A good user interface is easy to comprehend, ideally "walk up and use." Even a well written program, on the other hand, requires much more effort to understand.

4 Dr. Jones

DR. JONES is a planned program exploration tool for software archeology. Below, I will walk through a hypothetical session with DR. JONES to illustrate some proposed aspects of its user interface and functionality. A detailed discussion of the interface and an implementation framework await the completion of the exploratory user studies, which will provide guidance in choosing the kinds of program comprehension activities it should support.

4.1 Initial Steps: Structural Browsing

The archaeologist's session begins by starting DR. JONES and pointing it at a directory of source files. DR. JONES will parse the source to extract its abstract syntax tree, which forms the basis of structural views of the program. The archaeologist can immediately browse the software in terms of its syntax (packages, classes, methods, etc.) Structured documentation interspersed with the source code in the form of Javadoc comments is also extracted and used to annotate the structural views. Since program syntax is naturally hierarchical, this could be an appropriate place to use a multiscale diagram browser (such as the one in Figure 2).

Multiscale diagrams assist program comprehension in two ways. First, the archaeologist can see an overview of the program, and later choose which parts to examine in more detail. The archaeologist is not faced with all of the program's complexity at once. Second, the archaeologist can smoothly vary the level of detail. When the archaeologist wants to drill down to inspect a part of the program closely, she can do so by a mouse gesture (instead of navigating a filesystem). An example of this idea is shown in Figure 3, at left.

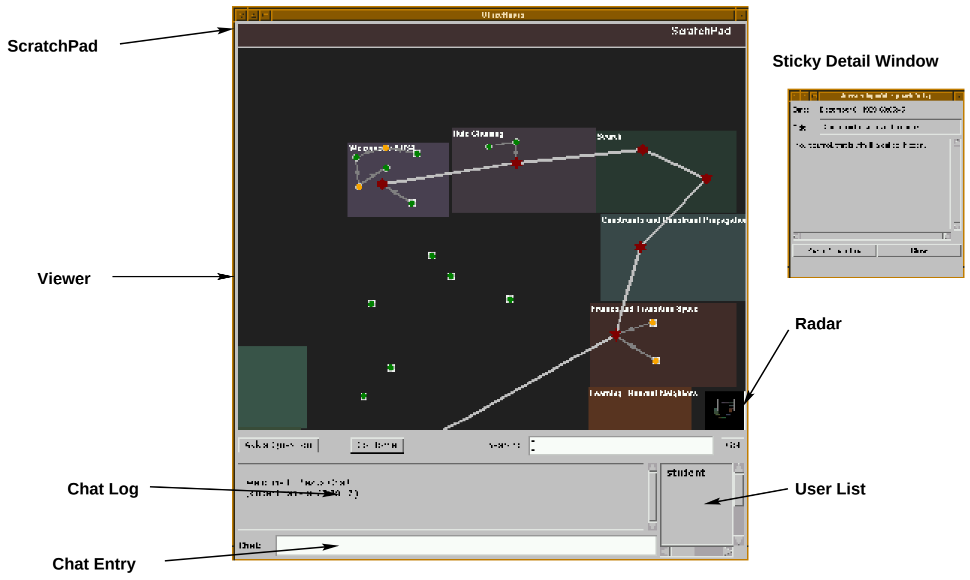

Figure 2: Plexus, a multiscale diagram browser, here used to visualize a knowledge base of questions and answers (Foltz *et al.*, 2000).

4.2 Pattern Recognition

Meanwhile, in the background, DR. JONES will attempt to recognize patterns in the program that correspond to meaningful abstractions and relationships. Examples include the multiplicity relations among classes, or subsystems that contain several related classes. When discovered, they will be added to the archaeologist's view of the program.

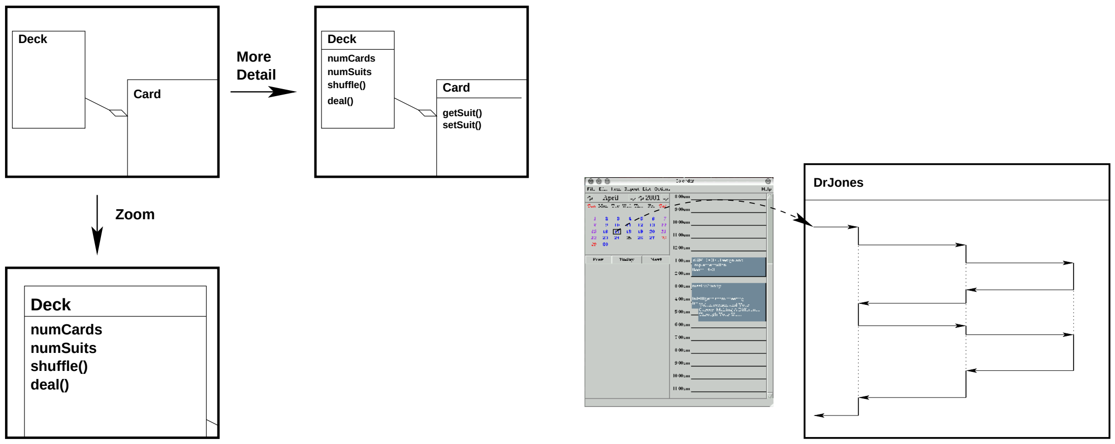

Figure 3: Interface ideas for DR. JONES. Left, adjusting the zoom and the level of detail. Right, "jiggling" software.

One purpose of these patterns is to introduce a hierarchy onto the structural view of the program, so that archaeologists can gain overviews and browse at an architectural level. These architectural features can derived by searching for object-oriented design patterns (Gamma *et al.*, 1995), domain-specific concepts, or by relating parts of the program to the archaeologist's own annotations (as discussed below).

4.3 Visualizing Behavior: "Software Jiggling"

The archaeologist now wishes to visualize the behavior of the program – the sequence of methods that are invoked in response to an external event (like a button press). One way to present this information is through an analogy to mechanical devices, whose behavior can be explored by jiggling its parts, and seeing what else moves in response. In a program, the archaeologist could gesture at a method and see where the control flow leads, or at a variable and see all of the reader and writer threads. Animation can also play a role, for example to show the movement of objects through the call stack.

For truly interactive exploration, DR. JONES could attach to a running Java virtual machine and allow the archaeologist to run the program in one window and see the execution trace resulting from user actions in another (Figure 3, right) – what "software jiggling" might look like.

4.4 The Archaeologist's Logbook

As the archaeologist makes discoveries, it would be convenient to have a place to record her findings. Alongside the software explorer could be a multimedia logbook, which could capture her notes and diagrams as the she browses the program. Ideally, this logbook lives "on top of" the views provided by DR. JONES, so that notes can be placed in spatial proximity to the relevant parts of the program. Such a diary should capture this new documentation in a natural and flexible manner, and be archived with the program so future archaeologists can benefit.

Also, as mentioned, an important part of program comprehension is building an analogy between the concepts of the application domain and the parts of the program. Since DR. JONES' views are built bottom-up, one place to incorporate domain-specific knowledge is to provide a medium for recording rough sketches of the domain model (including text and box-and-arrow diagrams). As the archaeologist discovers parts of the program that correspond to parts of the model, the two can be bidirectionally linked to record that fact.

4.5 Finishing Up

The archaeologist is finished when she feels she has gained a global understanding of the program, sufficient for her task. The end products of her session with DR. JONES would include:

- Diagrammatic views of program structure;
- Derived patterns and relationships in the program;
- Traces of program behavior, presented in terms of program structures;
- The archaeologist's own findings and notes, closely linked with the program text.

5 Assessment

Assessing the results of this research will be tricky – a program comprehension tool is very unlikely to help all programmers understand all programs. However, I plan to approach the problem by tackling the most obvious views and simple programs first, and then evolving the design of DR. JONES to handle more complicated cases. From our initial explorations, a view of program structure is a logical starting point.

The research will then proceed in multiple phases of exploratory user studies, development and prototyping of DR. JONES, and evaluation. The milestones for progress will center around three criteria: DR. JONES' ability to explain simple programs, favorable comparisons with Javadoc, and scaling results to large programs.

5.1 Explaining Simple Programs

The initial goal of DR. JONES is to produce reasonably complete and correct explanations of simple programs: for example, a calendar application that lets a user track his appointments. A small group of these programs will form a training set for design pattern recognition and be used to prototype the user interface. The initial milestone for the development of DR. JONES is meaningful and complete visualization of simple programs, which can be evaluated by inspection.

5.2 Comparison With Javadoc

The goal of DR. JONES is to help programmers build mental models of programs, which is usually done by reading documentation and browsing source. DR. JONES succeeds if programmers can understand programs more easily with it than with standard program documentation.

A baseline for comparison is Javadoc, the documentation tool provided in the Java Developer's Kit. Many Java programs are documented in this format, which annotates classes and methods with short textual descriptions. Javadoc is browsable as a hypertext to navigate the class hierarchy and look up class definitions.

A proposed comparison is to ask a subject to browse Javadoc or use DR. JONES to explore two unfamiliar programs with similar designs. After the exploration session, the quality of the mental models learned will be assessed. Also, the subject will be asked to subjectively describe her browsing strategies and experiences with the two tools.

5.3 Scaling up

The real need for a program comprehension tool is when the programmer is faced with a program so large it cannot be understood by browsing a few files. Programs with thousands or millions of lines of code, developed over many years by many programmers, are the real test cases for DR. JONES.

We do not expect DR. JONES to handle these cases as well as the simple calendar application. But they will still be worth trying, to find the breakdown points of the presentation metaphors, recognition system, and browsing interface. The hope is that the underlying mode of interaction, multiscale source browsing, will remain usable, but require additional knowledge to recognize the greater varieties of abstractions in programs of significant size and complexity.

Fortunately, the open source software movement has made available many examples of real, production code bases of varying size and design complexity. These examples will be valuable while incrementally improving the quality of design pattern recognition, as well as providing test cases for usability comparisons with Javadoc. Also, a useful exercise would be to analyze a production program whose developers are available in person to critique the views DR. JONES provides.

6 Conclusion

The thesis of this proposal is that there are a small number of tightly coupled program views that, taken together, can provide a programmer with an effective mental model of an unfamiliar program. Each of the views contains information programmers usually seek by browsing source code or reading good documentation. By presenting this information in diagrams, program comprehension can be transformed from a difficult, cognitive task into an easier, perceptual one (Larkin and Simon, 1987).

To support this thesis the research will study the strategies programmers use when they comprehend programs, to determine the kinds of information they extract and the design patterns they recognize. These findings will motivate how the program is viewed through DR. JONES, a multiscale program browser. DR. JONES acts as a software archaeologist's magic lens, making explicit patterns and relationships in the program that usually require much more effort to deduce.

6.1 Inspirations and Related Work

Research to develop of tools for program understanding and comprehension has been underway for some time. An early prototype of this work was the Programmer's Apprentice (Rich and Waters, 1989), an integrated, knowledge-based software engineering environment. It combined program understanding through design pattern recognition with intelligent assistance for design, optimization, and debugging. DR. JONES borrows the idea of design pattern recognition to seek abstractions above the class and method level.

The use of visualization for the explanation of complex devices and systems also has a rich history (Tufte, 1997). Figure 4 is an example of how a complex sequence of events can be communicated by illustrating significant interactions and suppressing irrelevant detail.

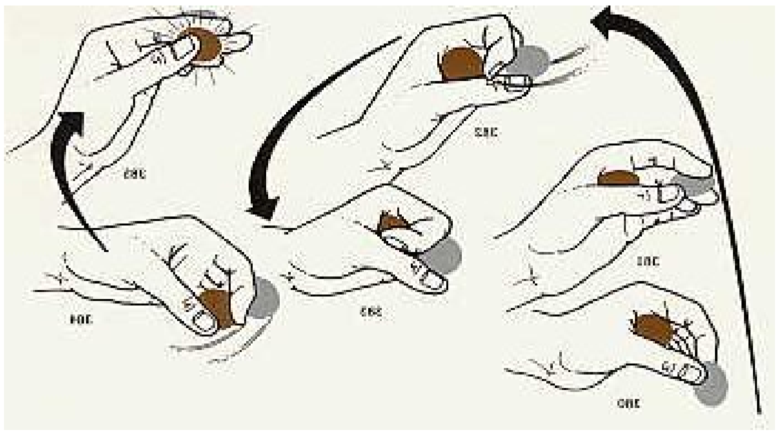

Figure 4: An example of a diagrammatic illustration of a complex sequence of events, from (Tufte, 1997).

Software visualization has been particularly successful for illustrating the behavior of algorithms (Stasko *et al.*, 1998). A visualization is usually designed for a particular family of algorithms (like sorting), limiting its usefulness for visualizing programs that use many types of algorithms. Also, the focus here is on program visualization at an architectural level, so that a method's function is much more relevant than its implementation. Software diagramming languages, like Universal Modeling Language (Booch *et al.*, 2001), are a related family of software visualizations and a source of ideas.

Comprehension tools that are cognitively motivated are most comparable to this proposal, such as DynaSee (Zayour and Lethbridge, 2000) and SHriMP (Storey *et al.*, 1997). To motivate DynaSee, the authors performed a cognitive analysis of program simulation in source code in terms of its load on working memory. DynaSee is designed to lighten this load by allowing the programmer to browse the execution trace as a collapsible tree. SHriMP, based on similar ideas, is a software structure browser that incorporates fisheye views to control the complexity of the display. DR. JONES will extend these approaches to develop a family of tightly coupled views at multiple levels of detail.

6.2 Surprises

At least two major kinds of surprises could alter the progress of this research. First, the exploratory user studies could uncover an unexpected variety of comprehension strategies; individual differences may dominate, and no discernible patterns would emerge. This will limit the influence of the user studies on the design of DR. JONES. If this is the case, an alternative is to borrow diagrams from UML and other diagramming languages. These have been proven in practice to be useful for program documentation, and motivate other work that seeks to provide a natural environment for software design by sketching (Hammond, 2001).

The second surprise is unexpected variation in the design and syntax of programs. Software is big and messy, and more often evolves through an accretion of features and workarounds than by planned design. This fact may limit the effectiveness of top-down abstraction based on the recognition of common design patterns. A bottom-up approach such as hierarchical clustering may turn out to be more reliable for the recognition of some kinds of abstractions.

An underlying assumption is that a method forms the atomic unit of functionality for comprehension. This assumption is violated when a programmer chooses poor abstractions and combines several loosely coupled functions into a single method. The identification of separable pieces of functionality within a method would help, but this is a difficult problem and beyond the scope of this proposal. Any program comprehension tool is ultimately limited by the design quality of the programs it examines – untangling spaghetti code remains, for now, the province of expert human programmers.

6.3 Prospects

Program comprehension is a critical part of software maintenance, and the day-to-day life of programmers would be much more pleasant if they had right tools to assist them at this task. In an ideal world, instead of expending valuable mental energy `grepping` through and poring over thousands of lines of code, the programmer will be able to interact with and visually explore the program, literally seeing how it works. If comprehension were removed as a bottleneck to software maintenance, the reality of software evolution could be altered: instead of reimplementing a system from scratch, as is often done, designers will be encouraged to adapt and reuse existing source code, which hopefully has most of its bugs worked out already.

While wandering around the hallways of the Lab, I've been jealous of the chip plots that VLSI designers post of their projects. A lot of information is immediately available, such as the location of the register file and cache, even without knowing much about the function of the chip (or, in my case, VLSI design). Although the comparison is not quite fair – chips, unlike software, never change after going to the foundry – a dream for DR. JONES is that it makes software structure and behavior just as accessible through its magic lens.

7 Appendix: Proposed Schedule of Research

Milestone/Deliverable	**Tentative Date**
User Studies: 10 Sessions	July, 2001
User Studies: 20 Sessions	August, 2001
User Studies: 30 Sessions	September, 2001
DR. JONES UI Low Fidelity Prototype	October, 2001
DR. JONES Infrastructure	November, 2001
DR. JONES UI Working Prototype	December, 2001
DR. JONES Small Programs Evaluation	January, 2002
DR. JONES Large Programs Evaluation	February, 2002
DR. JONES vs. Javadoc Evaluations (10 Sessions)	March, 2002
Thesis Document Draft	April, 2002
Thesis Defense	May, 2002

Note: A possible collaboration with Prof. Daniel Jackson, who teaches 6.170 in the Spring 2002 term, to provide an educational testbed for DR. JONES is not included.

References

Booch, G. *et al.* (2001). Universal Modeling Language specification version 1.3. Published by the Object Modeling Group.

Eick, S. G. (1998). Maintenance of large systems. In Stasko *et al.* (1998), chapter 21.

Erdos, K. and Sneed, H. M. (June 1998). Partial comprehension of complex programs. In *6th International Workshop on Program Comprehension (WPC'98)*, pages 98–105.

Foltz, M. A., Neviett, W. T., and Xiong, R. (2000). Architecting a group memory with Plexus. Manuscript.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, Massachusetts.

Hammond, T. (April 2001). Interpreting sketching in software development. Manuscript. MIT Artificial Intelligence Lab.

Joy, B., Steele, G., Gosling, J., and Bracha, G. (1998). *The Java Language Specification*. Second edition. Addison-Wesley.

Knuth, D. E. (1991). *Literate Programming*. Center for the Study of Language and Information, Stanford University. CSLI Lecture Notes Number 27.

Kramer, D. (1999). API documentation from source code comments: A case study of Javadoc. In *Proceedings of the Seventeenth Annual Conference on Computer Documentation*, pages 147–153.

Larkin, J. H. and Simon, H. A. (1987). Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, **11**(1):65–99.

Pauw, W. D., Kimmelman, D., and Vlissides, J. (1998). Visualizing object-oriented software execution. In Stasko *et al.* (1998), chapter 22, pages 329–346.

Raghavan, A. and Stahovich, T. F. (1998). Computing design rationales by interpreting simulations. In *Proceedings of Tenth International ASME Conference on Design Theory and Methodology*.

Rich, C. and Waters, R. C. (January 1989). Intelligent assistance for program recognition, design, and debugging. Technical Report MIT AI Memo 1100, MIT Artificial Intelligence Laboratory, Cambridge, MA.

Sayyad-Shirabad, J., Lethbridge, T., and Lyon, S. (May 1997). A little knowledge can go a long way toward program understanding. In *Fifth International Workshop on Program Comprehension*, pages 111–117. Institute for Electrical and Electronics Engineers.

Stasko, J., Domingue, J., Brown, M. H., and Price, B. A., editors (1998). *Software Visualization: Programming as a Multimedia Experience*. The MIT Press, Cambridge, MA.

Storey, M.-A. D., Fracchia, F. D., and Muller, H. A. (May 1997). Cognitive design elements to support the construction of a mental model during software visualization. In *Proceedings of the 5th International Workshop on Program Comprehension*, pages 17–28. Dearborn, Michigan.

Tufte, E. R. (1997). *Visual Explanations: Images and Quantities, Evidence and Narrative*. Graphics Press, Cheshire, Connecticut.

Zayour, I. and Lethbridge, T. C. (2000). A cognitive and user centric based approach for reverse engineering tool design. In *Proceedings of CASCON 2000*.