Cognitive Design Elements to Support the Construction of a Mental Model during Software Exploration

M.-A.D. Storey[†‡] F.D. Fracchia[‡] H.A. Müller[†]

[‡]School of Computing Science
Simon Fraser University
Burnaby, BC, Canada

[†]Department of Computer Science
University of Victoria
Victoria, BC, Canada

Abstract

The scope of software visualization tools which exist for the navigation, analysis and presentation of software information varies widely. One class of tools, which we refer to as **software exploration tools**, *provides graphical representations of static software structures linked to textual views of the program source code and documentation. This paper describes a hierarchy of cognitive issues which should be considered during the design of a software exploration tool. The hierarchy of cognitive design elements is derived through the examination of program comprehension cognitive models. Examples of how existing tools address each of these issues are provided. In addition, this paper demonstrates how these cognitive design elements may be applied to the design of an effective interface for software exploration.*

1 Introduction

It is widely accepted that time spent understanding existing programs is a significant proportion of the time required to maintain, debug and reuse existing code. Understanding programs is a very complex task. It is often especially difficult because program code may be the only source of information.

A variety of techniques and tools have been proposed to assist programmers in program comprehension. One of these techniques, *reverse engineering*, is the process of extracting high-level design information from source code. A reverse engineer uses these techniques to identify system components and their interrelationships and creates representations of the system in another form, usually at a higher-level of abstraction [1]. Tilley *et al.* identified three basic activity sets that are characteristic of the reverse engineering process [2]:

- *Data gathering* through static analysis of the code or through dynamic analysis of the executing program.
- *Knowledge organization* by organizing the raw data by creating abstractions for efficient storage and retrieval.
- *Information exploration* through navigation, analysis and presentation.

According to Tilley, information exploration "holds the key to program understanding". Some software visualization tools use graphical representations for the navigation, analysis and presentation of software information to further understanding [3]. For instance, several software visualization tools show animations of algorithms and data structures with the goal of teaching widely used algorithms and data structures. Another class of tools shows the dynamic execution of programs for debugging, profiling and for understanding run-time behavior. Other software visualization tools mainly focus on showing textual representations, some of which may be *pretty printed* to increase understanding [4, 5] or use hypertext in an effort to improve the navigability of the software [6]. Typography plays a significant role in the effectiveness of these textual visualizations.

One class of software visualization tools, which we refer to as *software exploration tools*, presents graphical representations of static software structures linked to textual views with the goal of helping a maintainer form a mental model of the software. Although the motivation for designing these tools may be obvious; how to design an *effective* tool is not so obvious. Many such tools already exist, but few of them are widely used in practice [3, 6]. In general, there has been very little empirical evaluation of these tools, with correspondingly little guidance on the desirable features of such a tool.

Many researchers have, however, studied how programmers understand programs through observation and experimentation. This research has resulted in the development of several cognitive theories to describe the comprehension process. Although the cognitive

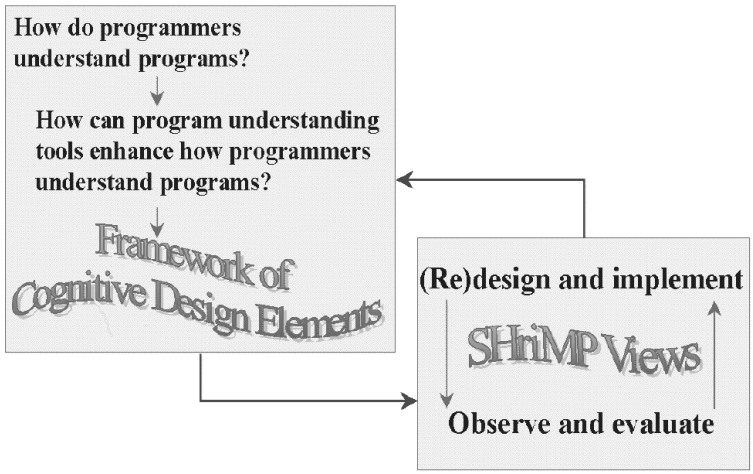

Figure 1: Exploration tool improvement cycle

theories differ in style and content, they share many elements and concepts which outline key activities in program understanding.

Software exploration tools are similar in many ways to hypermedia document browsers. A hypermedia document contains related and linked representations of an information space. Many of the difficulties experienced by a hyperdocument reader are the same difficulties as those experienced by the browser of a software visualization. Indeed, Thüring *et al.* [7] describe comprehension of a hyperdocument "as the construction of a mental model that represents the objects and semantic relations described in a text." They say that a document is *coherent* if a reader can construct a mental model which corresponds to something in the real world. In the context of software visualization, we could also say a visualization (or software documentation) is coherent if the maintainer can construct a mental model from the given visualization. A software visualization has *local coherence* when the maintainer can make sense of the statements and programming units and a visualization has *global coherence* if the maintainer can gain an understanding of the macrostructure of the program structure.

A hierarchy of cognitive issues for increasing the comprehension of hypermedia documents is described in [7]. Using this idea we have developed a related hierarchy to guide the development of a tool to aid in the exploration and comprehension of software systems. The hierarchy has two main branches. The first branch is intended to capture the essential processes of various strategies that programmers employ during comprehension. The second branch addresses the cognitive overhead experienced by a maintainer while browsing and navigating the visualization of the software structures. This second branch is similar to those issues which are also relevant for readers of hyperdocuments.

In an effort to describe the wide variety of software visualization tools, several taxonomies have been developed to classify these tools [3, 8, 9]. The most complete of these was described by Price *et al* in [3]. Their taxonomy is based on a generic model of a software visualization tool and uses categories such as *scope* (range of software that can be visualized), *content* (aspects to be visualized such as code, datatypes), *method* (how the visualization is specified), *interaction* (how the user interacts with the visualization), and *effectiveness* (whether the visualization fulfills its objectives). The cognitive framework described in this paper provides an alternative taxonomy for classifying existing software exploration tools. This taxonomy is based on the cognitive aspects of the maintainer rather than the concrete aspects of the tools themselves. The framework is explained in detail with references to a number of existing systems.

We are currently applying the framework of cognitive design elements to the design and evaluation of a tool for software exploration called SHriMP (**S**imple **H**ie**r**archi**c**al **M**ulti-**P**erspective) views [10]. We have performed two user studies to evaluate the effectiveness of the SHriMP interface [11, 12]. Results from these studies are being used to improve the cognitive framework for design. The improved framework will, in turn, be employed in the subsequent redesigns of the SHriMP tool (see Fig. 1).

The remainder of this paper is organized as follows. Section 2 describes several cognitive theories of program comprehension. Section 3 describes a hierarchy of cognitive design elements which should be considered when designing a software exploration tool. Examples of how existing software visualization tools address these issues are also provided. Section 4 discusses how the hierarchy of design elements is being applied to the design and evaluation of a software exploration tool. Section 5 concludes the paper.

2 Cognitive models of program comprehension

A *mental model* describes a maintainer's mental representation of the program to be understood. A *cognitive model* describes the cognitive processes and information structures used to form the mental model. Over the past 20 years, researchers have proposed many cognitive models to describe how programmers comprehend code during software maintenance and evolution. The following subsections describe some of the key cognitive models resulting from this research. All of these cognitive models rely on the maintainer's own knowledge together with the code and documentation to create a mental representation of the program [13].

2.1 Bottom-up program comprehension

Bottom-up theories of comprehension propose that understanding is built from the bottom up, by reading source code and then mentally *chunking* or grouping these statements into higher-level abstractions. These abstractions are aggregated further until a high-level understanding of the program is attained [14].

Shneiderman and Mayer's cognitive framework differentiates between syntactic and semantic knowledge of programs [15]. Syntactic knowledge is language dependent and concerns the statements and basic units in a program. Semantic knowledge is language independent and is built in progressive layers until a mental model is formed which describes the application domain. The final mental model is acquired through the chunking and aggregation of other semantic components and syntactic fragments of text.

Pennington's model [16] also has a bottom-up flavor. She investigated the role of programming knowledge and the nature of mental representations in program comprehension. She observed that programmers first develop a control-flow abstraction of the program which captures the sequence of operations in the program. This model is referred to as the *program model* and is developed through the chunking of microstructures in the text (statements, control constructs and relationships) into macrostructures (text structure abstractions or chunks) and by cross-referencing these structures. Once the program model has been fully assimilated, the *situation model* is developed. The situation model encompasses knowledge about data-flow abstractions (changes in the meaning or values of program objects) and functional abstractions (the program goal hierarchy). The development of the situation model requires knowledge of the application domain and is also built from the bottom-up.

2.2 Top-down program comprehension

Brooks [17] theorizes that programmers understand a completed program in a top-down manner where the comprehension process is one of reconstructing knowledge about the domain of the program and mapping that to the actual code itself. The process starts with a hypothesis concerning the global nature of the program. The initial hypothesis is refined in a hierarchical fashion by forming subsidiary hypotheses. The verification (or rejection) of hypotheses depends heavily on the absence or presence of *beacons* [17]. A beacon is a set of features that indicates the existence of hypothesized structures or operations.

Soloway and Ehrlich [18] also observed that top-down understanding is used when the code or type of code is familiar. Expert programmers use two types of programming knowledge during program comprehension [18]:

- *Programming plans* are generic fragments of code that represent typical scenarios in programming. For example, a sorting program will contain a loop which compares two numbers in each iteration.
- *Rules of programming discourse* capture the conventions of programming, such as coding standards and algorithm implementations.

According to Soloway and Ehrlich's observations, a mental model is built top-down by forming a hierarchy of goals and programming plans. Rules of discourse and beacons help to decompose goals and plans into lower-level plans.

2.3 Knowledge-based understanding model

Letovsky [19] views programmers as *opportunistic processors* capable of exploiting either bottom-up or top-down cues. There are three components to his model:

- The *knowledge base* encodes the programmer's expertise and background knowledge. The programmer's internal knowledge may consist of application and programming domain knowledge, program goals, a library of programming plans and rules of discourse.
- The *mental model* encodes the programmer's current understanding of the program. Initially it consists of a specification of the program goals and later evolves into a mental model which describes the implementation in terms of the data structures and algorithms used.
- The *assimilation process* describes how the mental model evolves using the programmer's knowledge base together with program source code and documentation. The assimilation process may be a bottom-up or top-down process depending on the programmer's initial knowledge base.

2.4 Systematic and as-needed program understanding strategies

Littman *et al.* [20] observed that either programmers *systematically* read the code in detail, tracing through the control-flow and data-flow abstractions in the program to gain a global understanding of the program, or they take an *as-needed* approach, focusing on only the code related to a particular task at hand. Soloway *et al.* [21] describe a model which merges the concepts of systematic strategies, as-needed strategies and inquiry episodes (as defined by Letovsky [19]) into a single model:

- *Micro-strategies* include inquiry episodes which consist of a *read, question, conjecture* and *search* cycle. Such episodes occur as a result of *delocalized plans*. A delocalized plan is conceptually related code located in non-contiguous parts of the program.
- *Macro-strategies* are used to achieve an understanding at a more global level. According to Soloway *et al.* there are two main macro-strategies:
  - *Systematic macro-strategies:* The programmer traces the flow of the entire program by reading all of the code and documentation, and performing simulations as they read. This strategy leads to more correct enhancements because causal interactions in the delocalized plans are discovered. However, it is unrealistic to systematically read all of the code for larger programs.
  - *As-needed macro-strategies:* The programmer studies only parts of the code that they think are relevant to the task at hand. More errors are made using this approach since causal interactions are often overlooked [21]. This approach is, however, the most common strategy.

2.5 An integrated metamodel of program comprehension

Von Mayrhauser and Vans' metamodel [13] integrates Soloway's top-down model with Pennington's program and situation models. During their experiments they observed some programmers frequently switching between all three comprehension models. The *Integrated Metamodel* consists of four major components. The first three components describe the comprehension processes used to create mental representations at various levels of abstraction and the fourth component describes the knowledge base needed to perform a comprehension process:

- The *top-down (domain) model* is usually invoked when the programming language or code is familiar. It incorporates domain knowledge which describes program functionality as a starting point for formulating hypotheses. The top down model is usually developed using an opportunistic or as-needed strategy.
- The *program model* may be invoked when the code and application is completely unfamiliar. The program model is a control-flow abstraction, and may be developed as an initial mental representation.
- The *situation model* describes data-flow and functional abstractions in the program. Pennington assumes that the situation model is developed only after the program model has been formed. Von Mayrhauser and Vans feel that this is unrealistic for larger programs [22]. In the integrated model, a situation model may be developed after a partial program model has been formed using systematic or opportunistic understanding strategies [23].
- The *knowledge base* consists of information needed to build these three cognitive models. It represents the programmer's initial knowledge before the maintenance task and is used to store new and inferred knowledge.

Understanding is formed at several levels of abstraction simultaneously by switching between the three

comprehension processes [23]. According to this model any of the three comprehension processes may be activated at any time [13]. This differs from Letovsky's model which states that comprehension occurs either top-down or bottom-up depending on the cues available.

2.6 Explaining the variation in program comprehension models

Although there are disparities in the comprehension models, these are due to the varied characteristics of the maintainer, program to be understood and the goal for comprehending the program. To understand how programmers understand programs, the factors that can effect the comprehension process must be considered.

Most researchers acknowledge that certain factors will influence the comprehension strategy adopted by a programmer. Vessey [24] states that we must control the factors which influence programmer performance. She specifically mentions program layout, language design, programming mode and programming support facilities. Brooks [17] noticed behavioral differences due to the problem domain, differences in program text, individual differences and the purpose for understanding the program. Von Mayrhauser and Vans [25] discriminate between the different strategies required for programs of varying sizes and different tasks. Lakhotia [26] noticed that the availability and validity of documentation had a strong impact on the comprehension strategy. Tilley *et al.* [2] describe how the experience and creativity of the maintainer will have an effect, as well as the quality, size and complexity of the program to be understood.

Table 1 summarizes the various factors which influence the comprehension process. These factors are due to differences among maintainers; differences in the program to be comprehended; and task differences. The comprehension models should, and many do, describe their model in the context of these characteristics. Researchers interested in studying programmer comprehension strive to limit factors which could influence their experiments, with the effect that their results are then dependent on the controlled factors. Even if these characteristics could be controlled in a laboratory experiment, these factors cannot be controlled in the real world.

A software exploration tool to aid in comprehension must help a maintainer in the key activities identified by the cognition model which best suits the given characteristics of the maintainer, program and task. It is unlikely that a single tool will be able to assist in all activities which are representative of the various cognition models. The next subsection describes a hierarchy of cognitive issues which should be considered when designing a tool to assist in the exploration of software structures.

3 Cognitive design elements for software exploration tools

Software exploration tools provide graphical representations of the software structure linked to textual representations of the source code and documentation with the goal of helping a maintainer form a mental model of the software. Of key importance is whether such a tool supports bottom-up comprehension, top-down comprehension or some combination of the two. Also important, especially for larger systems, is how the maintainer browses or navigates the visualization.

This section describes a hierarchy of cognitive design elements (**E1 – E14**) to guide the development of a tool to aid in the exploration and comprehension of software systems (see Fig. 2). This hierarchy has two main branches. The first branch is intended to capture the essential processes of the various comprehension strategies such as the top-down, bottom-up and integrated approaches. The second branch addresses cognitive overhead experienced by a maintainer browsing and navigating a visualization of the software structure.

3.1 Improve program comprehension

Since the comprehension strategy employed by a maintainer is dependent on a variety of factors dictated by the maintainer, program and task, it would be advantageous for a tool to support a wide array of comprehension activities. Usually specialized tools to suit a particular comprehension strategy are developed which may result in simpler, easier to use tools. This section further explores the comprehension models presented in Section 2 and extracts cognitive design elements which should be addressed by a tool claiming to aid a given comprehension strategy.

3.1.1 Enhance bottom-up comprehension

Bottom-up comprehension involves reading program statements and constructs and chunking these units into higher-level abstractions, until an overall understanding of the program is attained. Bottom-up comprehension involves three main activities: 1) identifying software objects and the relations between them; 2) browsing code in delocalized plans; and 3) building

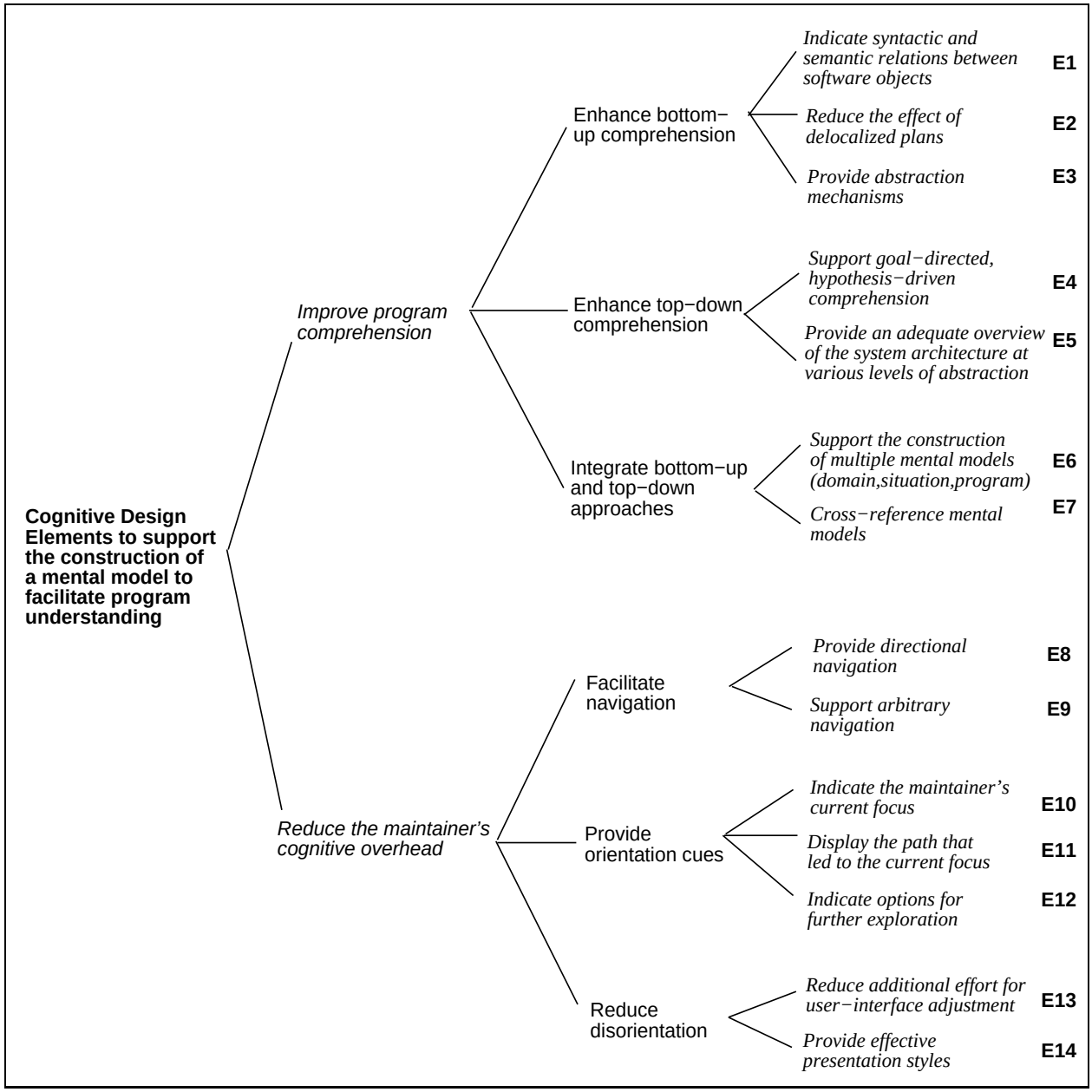

Figure 2: Cognitive Design Elements for Software Exploration

Table 1: Influences on Program Comprehension Strategies

Maintainer Characteristics	Program Characteristics	Task Characteristics
application domain knowledge	application domain	task type, purpose
programming domain knowledge	programming domain	task size and complexity
maintainer expertise, creativity	program size, complexity, quality	time and cost constraints
familiarity with program	documentation availability	environmental factors
support tools expertise	support tool availability	

abstractions (through chunking) from lower-level units. A comprehension tool to assist in bottom-up comprehension should address these main activities.

E1: ***Indicate syntactic and semantic relations between software objects***

A software visualization should provide immediate and visible access to the lowest level units in a program such as the code or visual icons representing these atomic units. The syntactic and semantic relations of these units must be clearly visible and easily accessible. The syntactical relationships between these units describe the text-structure at the micro-structure and macro-structure levels. These relationships are easily derived from source code listings. Semantic relations between software objects require data-flow or functional knowledge of the program. Many tools present this information in the form of a graph where nodes represent software objects and arcs show the relations between the objects. This method is used by PECAN [27], Rigi [28], VIFOR [29], Whorf [30], CARE [31], Hy+ [32], Imagix 4D [33] among others. In some systems, direct links from the software objects to the corresponding source code are also provided.

E2: ***Reduce the effect of delocalized plans***

A delocalized plan results from the fragmentation of source code related to a particular algorithm or plan. Without tool assistance, reading code belonging to a delocalized plan can be cumbersome as it may involve frequent switching between files and result in a feeling of disorientation.

Whorf [30] was specifically designed to reduce the effects of delocalized plans. It supports multiple views of the program such as source code listings, call-graphs, variable cross-reference and function cross-reference views. Views are linked by displaying different instances of an object using the same color in each of the views. Code in delocalized plans is highlighted reducing the effects of fragmentation.

Static analysis tools, such as *program slicing*, can identify code belonging to a delocalized plan. Program slicing is a method for decomposing a program into components where each component describes some of the system's functionality. A program slice contains all of the code which is relevant to that behavior [34]. SeeSlice [35] is a tool for visualizing program slices where program files are displayed as columns that contain line representations of procedures. Code that is not part of a slice is elided. Ghinsu [36], a toolset for program understanding, displays slicing results in its system dependence graph to capture the control and data dependencies in the software. The developers of Ghinsu recognize that non-local interactions in the code are a major cause of complexity, and so their toolset specifically addresses this problem.

E3: ***Provide abstraction mechanisms***

The process of building mental hierarchical abstractions from the low-level software objects and relations is the hardest part of bottom-up comprehension for many maintainers, and yet many tools only support showing a previously abstracted view [37]. Maintainers might understand the software better through abstractions they created themselves, rather than through the prefabricated and less trusted abstractions that many tools provide. Facilities should be available to allow the maintainer to create their own abstractions and label and document them to reflect their meaning.

Abstraction can be supported by selecting lower level objects and aggregating them into higher-level abstractions. In several visualization tools, a subgraph (a set of nodes and arcs) may be *collapsed* into a single *composite* or *subsystem* node [33, 38, 39, 40]. The same functionality is available in CARE [41] using the *compose* command. Both CIA [42] and Rigi [38] facilitate abstraction through *subsystem identification*. A subsystem is a collection of related nodes and arcs in the software graph. For example, a subsystem may include a function with all of the functions and datatypes it directly or indirectly depends on; such a subsystem is referred to as a *compilable slice* in CIA. Other subsys-

tems may be identified automatically by analyzing the binding strength between a pair of software objects. For example, two objects which both reference many other objects would have common clients or suppliers [42]. In Rigi, subsystem identification techniques are end-user programmable using the RCL command language [43]. In this way subsystem identification can be semi-automated by leveraging application or programming domain knowledge. For example, all nodes labeled with a common prefix according to some naming convention can be collapsed into a single subsystem [44]. The paper by Kimelman *et al.* [39] describes using arcs to represent subgraphs, in addition to using nodes for abstraction [39].

Several tools provide the ability to filter visually (temporarily hide) objects which results in a less detailed or abstract graph. Rigi [38], Imagix [33] and CARE [41] all provide filtering mechanisms. In HierNet [40] nodes represent source files and modules, and arcs represent simultaneous change requests to the connected nodes. Arcs are assigned a weight to show the number of change requests. Nodes and arcs can be then filtered using a threshold on the weights of the arcs. In SeeSlice [35] hierarchies shown contain three levels of abstraction: files (modules), procedures and statements. The user can filter any of these three levels of abstraction using a slider. VIFOR supports filtering so that only a subset of the database need be displayed at a time [29]. Nodes which represent dead code (functions not called directly or indirectly) can be elided in CIA [42] and in Rigi [45].

3.1.2 Enhance top-down comprehension

Understanding a program top-down requires application domain knowledge, previous exposure to the program or access to documentation describing the program design and evolution history. The maintainer formulates hypotheses and reads the code in a depth-first manner to verify or reject these hypotheses. A tool supports this process by providing a method for documenting hypotheses and linking the hypotheses to relevant parts of the program code or documentation. Facilities to refine hypotheses into subsidiary hypotheses must also be provided. Alternatively, the tool may provide a layered view of the program (previously prepared during system evolution or through reverse engineering) which entices a maintainer to explore the program in a top-down fashion.

E4: ***Support goal-directed, hypothesis-driven comprehension***

Relatively few systems facilitate top-down comprehension, where the programmer has an initial mental model or hypothesis concerning the functionality of the program. The TLES system (Tool for Layered Explanation of Software) is compatible with the top-down theory of software understanding and supports the creation of a chain of hypotheses and subsidiary hypotheses concerning the properties of the code [46]. The tool records these hypotheses for future maintenance. All information needed for understanding is stored in layers of annotations for recording the evolutionary history of source code constructs. This mechanism also supports recording of postponed or discarded hypotheses, which may be useful documentation for future maintenance [22].

Hy+ provides some support for verifying hypotheses concerning the design of the program [32]. It is a general purpose tool and has been used for querying and visualizing object-oriented systems. Hy+ uses a visual query language called Graphlog for querying the database. The results of a query are then used as a basis for the graphical views. This tool is useful for identifying *design patterns* [47] in the code. Design patterns are high-level design descriptions in object oriented code. Using Hy+, a maintainer can hypothesize that a particular design pattern exists and then search for it in the code by querying the database.

E5: ***Provide overviews of the system architecture at various levels of abstraction***

To explore programs top-down, access to the software architecture should be provided at various levels of abstraction. In Rigi, a software engineer or reverse engineer documents a program from the bottom-up by creating a hierarchy of abstractions. This hierarchy is then available for top-down exploration during subsequent maintenance. To facilitate access to the program at different levels of abstraction, Rigi supports *overview windows* which show the hierarchical structure of the software structure and *general windows* which contain slices of the hierarchy at selected levels of abstraction [48].

Landscape views [49], Hy+ [50], SHriMP views [48] and Continuous Zoom [51] all use a *nested graph* representation of software architecture. The hierarchical structure is displayed by the nested graph. Information at any level of information can be displayed or elided to show overviews of the system architecture at selected levels of abstraction.

Seela [52], a reverse engineering tool, converts code into a program design language which the user can then edit in the form of structure charts. In this way high-

level documentation is generated describing the code structure. It supports the top-down comprehension process, as the source code is analyzed and then displayed on the screen so that it appears as a readable program-design. Program blocks can be renamed by the user, and the names are subsequently only shown in certain views. It provides top-down documentation, but this documentation is also built from the bottom-up.

3.1.3 Integrate bottom-up and top-down approaches

Von Mayrhauser and Vans observed programmers using both bottom-up and top-down approaches [22]. Programmers create various mental models and frequently switch between them during the course of comprehension. The program model describes the control-flow abstractions of the program. The situation model describes the data-flow and functional abstractions. Both control-flow and data-flow mental models can be presented visually using a graph. Displaying how functional abstractions relate to the application domain is a harder task. Another model which can be visually presented shows the behavior of an executing program. A tool addressing the integrated comprehension process should support the construction of several linked views representing a variety of cross-referenced mental models.

E6: ***Support the construction of multiple mental models***

Not only do mental models differ in context and level of abstraction, but they also differ from one maintainer to another [53]. Several mental models of a program may be presented visually using multiple views. Many of the tools already mentioned support multiple views of textual and graphical views [27, 29, 30, 31, 33, 35, 45, 54]. Example graphical views show call-graphs and variable usage diagrams. Example textual views include displaying source code, requirements documents, design specifications, program slices and software statistics (metrics). Some tools, such as PECAN [27], also support views showing the execution of the program (e.g., allocation and deallocation of heap data). Multiple views are often shown side by side, or displayed using overlapping, cascading or scrollable windows. Von Mayrhauser and Vans note that many tools support recording information for the mental model at the program level, but few tools support recording information for the situation and domain models [22].

E7: ***Cross-reference mental models***

Maintainers frequently switch from one model to another in the course of comprehension [23]. Often these switches are the result of a maintainer mentally cross-referencing different mental models. These mental models should be linked to record the cross-referencing of information for later use. In some systems, multiple views are visually linked by highlighting instances of the same object in all views. This is how multiple views are linked in PECAN [27], Whorf [30], CARE [31], Imagix [33] and Rigi [45]. Many systems also support synchronized views by updating all views when one view is altered in some way. For example, Hy+ supports synchronized graphical and textual browsing of source code [32].

The Programmer's Apprentice tool has direct support for both development of the program model and situation model [55]. This tool uses *Plan Calculus* to formally represent programs and plans. Plan Calculus has a graphical notation and a formal semantics which can be used to show a mapping between an abstraction of its implementation (the program model) and a specification abstraction (the situation model). A diagram for each model is displayed side by side with hooked lines to indicate correspondences between the two diagrams.

Several tools make use of sophisticated graphics techniques to display a third dimension on the screen. The prototype system, VOGUE, uses the third dimension for monitoring the performance of parallel/concurrent programs [56]. Their 3D framework displays one relation in two dimensions and assigns another meaning to the third axis. One mental model or view can be shown in the xy plane, and another mental model is shown in the yz plane. By rotating through the third dimension, the user can mentally integrate two separate mental models. Although this technique has potential, there are several unresolved user interface issues of how to manipulate the two views. PLUM [57] and the Durham 3D visualization tool [58] also make use of a third dimension to cross-reference multiple hierarchies.

3.2 Reduce the maintainer's cognitive overhead

When comprehending larger software systems, cognitive overhead increases rapidly. Visualization tools are often supplied in an effort to reduce cognitive overhead. Cognitive overhead can be alleviated by providing good navigation facilities, meaningful orientation cues, and by effectively presenting the information so

that it can contribute to program comprehension. Navigation provides the facilities to go from one place to another. Orientation cues show the user where they are currently, how they got there and how to go somewhere else [7].

Although a tool may provide many navigation methods and effective orientation cues, the user may still feel overwhelmed if presented with too much information. Effective presentation techniques can alleviate the effects of displaying large amounts of information [59]. Disorientation may also result from a badly designed user interface which lacks in a feeling of continuity between displays [7].

3.2.1 Facilitate navigation

Navigation facilities include mechanisms for browsing source code, program documentation, graphical views of software structure and documented mental models of the program.

E8: ***Provide directional navigation***

Directional navigation describes mechanisms for reading source code and program documentation sequentially, browsing the software using data-flow and control-flow relationships, traversing software structure in hierarchical abstractions and by following user-defined program or application dependent links. Source code and documentation may be browsed sequentially using text editors or by following control-flow or data-flow paths by linking nodes and arcs in graphical representations to the corresponding source code. Alternatively, code and documentation may be navigated using hypertext links. In Imagix [33], code and documentation are generated in HTML format to be viewed by a web browser. Subsystem hierarchies are navigated in Whorf [30], CARE [31], Imagix [33] and Rigi [45] by opening a window to show a view of the subsystem node selected.

E9: ***Support arbitrary navigation***

Arbitrary navigation is supported when a maintainer navigates to locations not necessarily reachable by following application or user-defined links. Arbitrary navigation is supported in books by readers *dog-earing* the corners of pages, and in hypermedia documents by symbolically marking pages of interest. Few tools (other than tools which provide hypertext like access to source code and documentation) provide this form of navigation access. Saving views (supported in PECAN [27] and Rigi [45]) may be used as a mechanism for storing arbitrary navigation steps: a maintainer may create a snapshot of the current view so that it may be immediately accessed in the future without having to follow defined links in the software visualization. Plaisant *et al.* mention the importance of being able to save points for rapid return and automatic navigation for information exploration [60]. Searching capabilities are available in several tools to provide another mechanism for arbitrary navigation [28, 30, 31, 33].

Navigation among the various mental models is the key to successfully using them for comprehension [22]. This is a non-trivial problem, as there may be one-to-many and many-to-one links from one model to another. For example, a one-to-many mapping occurs when a description of some of the program functionality pertains to many chunks of code in several source files. Some tools show mappings between two views visually. However, this approach requires that both models are displayed concurrently on the screen which may not be feasible for larger software systems. The application of a third dimension for showing an alternative view (as in VOGUE [56] and PLUM [57]) provides a navigation mechanism between two views as they can be navigated by rotating from one view to another. This is an interesting approach but there is the extra overhead of having to display and navigate in three dimensions on a two-dimensional surface.

3.2.2 Provide orientation cues

Orientation cues indicate to the maintainer where they are currently exploring in the software structure, how and why they are there and how to switch to a different focus.

E10: ***Indicate the maintainer's current focus***

Depending on the task at hand, a maintainer may be interested in viewing source code for a function, examining a diagram which describes some of the program's functionality or browsing a set of documentation. The focus of interest may be fragmented as the maintainer tries to understand non-local interactions in the code. The use of judicious orientation cues can be used to indicate the current focus in a complex display.

Indicating the maintainer's current focus, requires not only showing the artifacts that are of immediate interest, but also displaying context for those artifacts. Textual views of source code implicitly show the focus since the code of interest is directly visible. However, other related code may not be visible. Many systems, such as Rigi [45] and Whorf [30], use highlighted nodes and arcs to emphasize the current focus in a graph but in larger graphs highlighted nodes and arcs may not

always be obvious.

Some software visualization systems (Hy+ [32] SHriMP Views [48], Continuous Zoom [51] and PLUM [57]) make use of *fisheye* display techniques [61] which allocate more screen space to more important information by displaying it larger than secondary information. Both PLUM [57] and GraphVisualizer3D [62] use the third dimension where more important nodes are drawn closer to the user's view point and appear larger due to perspective projection. VIPR, a scalable interface for visualizing Tcl programs [63], provides zooming and fisheye view methods where nested circles are used to represent program constructs. An execution view shows dynamic execution for debugging purposes. The code currently being executed is drawn larger using their fisheye view technique.

E11: ***Show the path that led to the current focus***

In graph representations of software structures, nodes and arcs are often used to access other parts of the software. Accessed nodes in the graph may be highlighted in an overview window to show the path traveled in the software hierarchy. Recording why a maintainer is interested in a particular software object is very important.

In hypermedia document browsers there are often *histories* or *breadcrumb trails* of traveled paths to indicate to the reader how a particular document in the structure was reached. Similar facilities would be useful when browsing HTML'ized software documents. The reason for reading a piece of code may be the result of verifying a particular hypothesis or because the code must be changed or adapted in some way. There is typically little tool support for recording this sort of temporary information.

E12: ***Indicate options for further exploration***

Given that a user is at a certain point in the exploration of a software system, this design element addresses not which facilities are available for further exploration, but rather how the user is made aware of the facilities available for further exploration. In textual views, a maintainer can browse related code by opening other source files explicitly. Some tools provide HTML views of the source code and documentation [33, 64]. Web browsers are used to browse related code using hyperlinks. The hyperlinks are the visual cues for accessing other parts of the documentation. The EDSA (Expert Dataflow and Static Analysis) tool [52] allows a maintainer to follow data-flow or control-flow paths in program slices. In graphical representations of software structure, the graph itself can be used to display further navigation options.

3.2.3 Reduce disorientation

For the exploration of larger systems, reducing disorientation effects is critical. Users may feel disoriented when scrolling through views of large systems, or when navigating amongst numerous windows. Disorientation can be alleviated by removing some of the unnecessary cognitive overhead resulting from poorly designed user interfaces and by using specialized graphical views for presenting large amounts of information.

E13: ***Reduce additional effort for user-interface adjustment***

Poorly designed interfaces will of course induce extra overhead. Available functionality should be visible and relevant [65] and should not impede the more cognitively challenging task of understanding a program.

Significant cognitive overhead may be introduced due to the disorientation caused by switching views for different mental models. SeeSys [66] provides a slider which the maintainer can use to animate the views with respect to time. However, there is a discontinuity between the views which may cause disorientation. Kimelman *et al.* describe the application of morphing techniques to iterate smoothly between different layouts [39]. Although there is extra overhead involved in computing the transitions between views, the effects of reduced disorientation may be worth the additional effort. The cognitive overhead of switching between views at various levels of detail can be alleviated by animating zoom-in (enlarge) and zoom-out (shrink) actions. This functionality is supported in SHriMP Views [48], Continuous Zoom [51] and VIPR [63].

E14: ***Provide effective presentation styles***

For complex graphs typical of larger software systems, layout algorithms are frequently used to display the graph in a more meaningful manner. Although software has no inherent shape or color, a graph can be drawn in such a way that it communicates key characteristics about the software. For example, a graph which contains many crossing arcs may give the impression of increased complexity in the software. Many software visualization tools recognize the importance of graph layouts and provide specialized or customized layouts suitable for presenting software graphs [29, 31, 42, 43, 40, 50, 57, 66].

4 Towards an effective interface for software exploration

We are currently applying the cognitive design elements developed in Section 3 to the design of the SHriMP software exploration tool [10]. This approach was developed in response to some deficiencies identified with the visualization methods used in the Rigi reverse engineering system [10]. The Rigi interface consists of a multiple window approach for displaying software structures. It has been observed that some users frequently lose context due to a lack of orientation cues. In addition, users have difficulties switching from one view to another when new windows are opened.

The SHriMP technique uses a fisheye view of nested graphs to show a single view of the software structure (see Fig. 3). The nested graph view has *composite nodes* that contain other nodes, forming a hierarchical structure. Composite nodes typically represent software subsystems (perhaps discovered during reverse engineering). *Composite arcs* represent one or more arcs between lower-level nodes in the hierarchy. Composite arcs can be selected and opened to display the lower-level, constituent arcs they represent. The nested graph display of software structures is useful for displaying multiple levels of abstraction and provides effective orientation cues which are lacking in the multiple window interface used in Rigi.

SHriMP integrates fisheye and pan+zoom approaches for magnifying nodes of interest [67]. Figure 3 shows a fisheye view of the GamePlay subsystem in the Hangman program. The GamePlay subsystem has been magnified to show more detail while the rest of the graph was shrunk to allocate more space to the GamePlay node. The interactive fisheye view may reduce cognitive overhead for user interface adjustment since it shows both context and detail in a single view. However, depending on the given task and required information, contextual cues may not always be needed. A traditional pan+zoom approach integrated in SHriMP allows the user to pan and zoom around a single view. To show more detail for a node of interest, the user selects the node and zooms in until the required level of detail is visible.

SHriMP is specifically being tailored to address issues relevant to the integrated model of program comprehension. Support for switching between mental models at various levels of abstraction has been added. A recent refinement (described in [67]) embeds HTML'ized source code fragments within the leaf nodes of the nested graph view. Each leaf node directly corresponds to a chunk of code in the program. SHriMP integrates the hypertext link-following metaphor (at the code level) with animated panning and zooming motions over the nested graph (at the structural level). Consequently, following a link to another function pans and zooms the view so that this function's code is presented within its node.

The first prototype of the SHriMP interface [48] was implemented in Tcl/Tk [68]. Tcl/Tk is a scripting language and user interface library useful for rapidly prototyping graphical interfaces. However, its graphics capabilities are not optimized for efficiently displaying the large graphs typical of software systems. The second prototype has been implemented using Pad++ [69], a graphics extension for Tcl/Tk. Pad++ is highly optimized for efficiently displaying large numbers of objects and smoothly animating the motions of panning and zooming.

We have performed two user studies to evaluate the effectiveness of the SHriMP interface [11, 12]. In the second experiment, 30 subjects (graduate and undergraduate computer science students) were observed while solving some realistic software maintenance tasks. In this study, SHriMP was compared to two other tools, Rigi and SNiFF+[1]. We hypothesized that a given interface would affect the comprehension strategy adopted by a subject to complete an assigned task. For SNiFF+, we expected that its predominantly textual interface would encourage a bottom-up comprehension strategy, where the code is read in some detail before drawing any conclusions about the global program structure. For Rigi, we expected that it would promote a top-down strategy, where the global subsystem structure would be understood before browsing the code in detail. Finally, we expected that the SHriMP interface would support frequent switching between top-down and bottom-up approaches due to the seamless integration of source code and high-level graphical views. Our observations do seem to support this hypothesis, but further studies are needed to verify this conjecture.

We intend to use the hierarchy of design elements as a blueprint against which the distinctive features of SHriMP and other software exploration tools may be compared from a cognitive perspective. Table 2 lists features in SHriMP which were added to provide support for the design elements outlined in the cognitive framework. Through experimentation and observation, we will study if the prescribed features adequately support the corresponding cognitive design elements. In particular, we plan to study the effects of various influential factors, such as maintainer expertise, program-

[1] SNiFF+ is a software development environment with reverse engineering and program understanding capabilities [70].

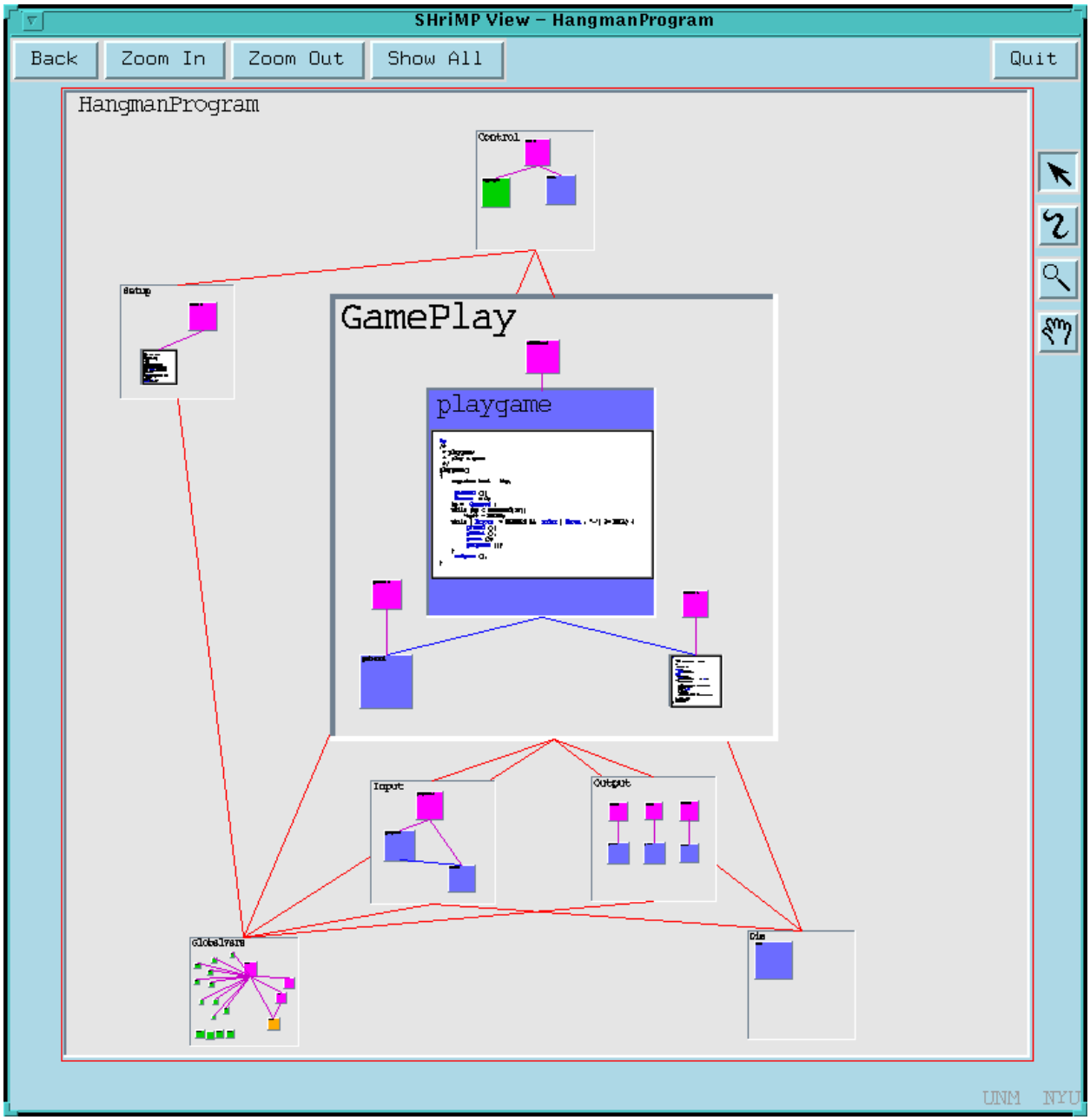

Figure 3: A SHriMP View of a program which implements a Hangman game. The main subsystems (Control, Setup, GamePlay, Input, Output, GlobalVars and Die) are shown in this view. A fisheye view of the GamePlay subsystem provides more detail since it is shown larger than the other subsystems. The maintainer can browse the source code by following hyperlinks within an architectural view of the entire program.

Cognitive Design Element	Corresponding feature in SHriMP
(*Enhance bottom-up comprehension*)	
E1: Indicate syntactic and semantic relationships	Source code view; graph nodes and arcs
E2: Reduce the effects of delocalized plans	Hypertext links in source code; arcs in the graph
E3: Provide abstraction mechanisms	Subsystem nodes and composite arcs in the nested graph
(*Enhance top-down comprehension*)	
E4: Support hypothesis-driven comprehension	Subsystem nodes can be annotated as they are created
E5: Provide overviews at various levels of abstraction	Subsystem nodes can be closed to show more abstract views
(*Integrate bottom-up and top-down approaches*)	
E6: Provide views of multiple mental models	Source code views; higher-level graphical views
E7: Cross-reference multiple mental models	Integration of source code view in graphical views
(*Facilitate navigation*)	
E8: Provide directional navigation	Hypertext links; navigation of subsystem nodes
E9: Provide arbitrary navigation	Search tool; hypertext browser back button; saving views
(*Provide orientation cues*)	
E10: Indicate the current focus	Fisheye views and detailed views of nodes of interest
E11: Display path that led to current focus	Nested graph
E12: Indicate options for further exploration	Hypertext links; nodes in the graph; thumbnail images
(*Reduce disorientation effects*)	
E13: Reduce effort for user-interface adjustment	Animation between views; fewer windows
E14: Provide effective presentation styles	Graph layouts; nested graphs; filtering options

Table 2: Designing the SHriMP tool using the Cognitive Framework for Design

ming domain and task purpose, on programmer strategy combined with the use of the SHriMP tool.

5 Conclusions

On review of the literature, there are several issues pertinent to program comprehension which are not adequately addressed by current software exploration tools. Although many tools do support bottom-up comprehension, relatively few tools support the integrated and top-down comprehension models. In particular, more support for mapping domain knowledge to code and switching between mental models would be useful. Better navigation methods which encompass meaningful orientation cues and effective presentation styles for browsing large software systems are also needed.

In general there has been a lack of evaluation of software exploration tools. Notable exceptions are [11, 12, 30, 41, 46, 71]. Despite the large number of software visualization tools, few of these tools are used in practice. More empirical studies of programmers are needed to help ascertain what kinds of tools are needed and which features would enhance the task of program comprehension.

In this paper a set of cognitive design issues was identified by examining various cognitive models of program comprehension and examples of existing software exploration tools. This framework of cognitive design issues is intended to be used as a guide during the design of software exploration tools. The framework by itself does not provide sufficient insight as to how the identified issues should be resolved, nor does it prescribe when or if certain information (such as histories of program changes [72]) should be made available to the maintainer. It does, however, highlight issues which may be important concerns when designing and evaluating these tools. In addition, the framework can be used to classify existing tools.

We are currently planning further user studies to evaluate the SHriMP and Rigi interfaces. Observations from these studies will be used to refine and strengthen the framework of cognitive design issues which will, in turn, be used to improve subsequent redesigns of the SHriMP interface. Using this iterative cycle of design and test, we are working towards a more effective tool for software exploration.

Acknowledgments

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada, the University of Victoria, and Simon Fraser University. The authors thank Kenny Wong and the anonymous reviewers for their helpful comments.

References

[1] E.J. Chikofsky and J.H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pages 13–17, January 1990.

[2] S.R. Tilley, S. Paul, and D.B. Smith. Towards a framework for program understanding. In *4th Workshop on Program Comprehension (WPC'96),* Berlin, Germany, pages 19–28, March, 1996.

[3] B. A. Price, R. M. Baecker, and I. S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, pages 211–266, June 1993.

[4] R.M. Baecker and A. Marcus. *Human Factors and Typography for More Readable Programs.* ACM Press, Addison-Wesley Publishing Company, 1990.

[5] T.D. Hendrix, J.H. Cross II, L.A. Barowski, and K.S. Mathias. Tool support for reverse engineering multilingual software. In *Proceedings of the 4th Working Conference on Reverse Engineering (WCRE'97),* Amsterdam, The Netherlands, pages 136–143, October, 1997.

[6] M. Petre, A.F. Blackwell, and T.R.G. Green. Cognitive questions in software visualization. In *Software Visualization: Programming as a Multi-Media Experience.* MIT Press, 1997. To appear.

[7] M. Thüring, J. Hannemann, and J.M. Haake. Hypermedia and cognition: Designing for comprehension. *Communications of the ACM*, 38(8):57–66, August, 1995.

[8] Brad A. Myers. Visual programming, programming by example, and program visualization: A taxonomy. In *CHI '86: Human Factors in Computing Systems*, pages 55–66, 1986.

[9] Gruia-Catalin Roman and Kenneth C. Cox. A taxonomy of program visualization systems. *IEEE Computer*, 26(12):11–24, December 1993.

[10] M.-A. D. Storey and H.A. Müller. Manipulating and documenting software structures using SHriMP views. In *Proceedings of the 1995 International Conference on Software Maintenance (ICSM '95)* (Opio (Nice), France, October 16-20, 1995), 1995.

[11] M.-A.D. Storey, K. Wong, P. Fong, D. Hooper, K. Hopkins, and H.A. Müller. On designing an experiment to evaluate a reverse engineering tool. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE'96),* Monterey, California, pages 31–40, November, 1996.

[12] M.-A.D. Storey, K. Wong, and H.A. Müller. How do program understanding tools affect how programmers understand programs? In *Proceedings of the 4th Working Conference on Reverse Engineering (WCRE'97),* Amsterdam, The Netherlands, pages 12–23, October, 1997.

[13] A. von Mayrhauser and A.M. Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, pages 44–55, August 1995.

[14] B. Shneiderman. *Software Psychology: Human Factors in Computer and Information Systems.* Winthrop Publishers, Inc., 1980.

[15] B. Shneiderman and R. Mayer. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer and Information Sciences*, 8(3):219–238, 1979.

[16] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.

[17] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–554, 1983.

[18] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10(5):595–609, September, 1984.

[19] S. Letovsky. Cognitive processes in program comprehension. In *Empirical Studies of Programmers*, pages 58–79. Ablex Publishing Corporation, 1986.

[20] D.C. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. In *Empirical Studies of Programmers*, pages 80–98. Ablex Publishing Corporation, 1986.

[21] E. Soloway, J. Pinto, S. Letovsky, D. Littman, and R. Lampert. Designing documentation to compensate for delocalized plans. *Communications of the ACM*, 31(11):1259–1267, 1988.

[22] A. von Mayrhauser and A.M. Vans. From code understanding needs to reverse engineering tool capabilities. In *Proceedings of CASE'93,* Singapore, pages 230–239, July, 1993.

[23] A. von Mayrhauser and A.M. Vans. Comprehension processes during large scale maintenance. In *Proceedings of the 16th International Conference on Software Engineering,* Sorrento, Italy, pages 39–48, May 1994.

[24] I. Vessey. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, 23:459–494, 1985.

[25] A. von Mayrhauser and A.M. Vans. Dynamic code cognition behaviors for large scale code. In *Workshop on Program Comprehension (WPC'93)*, pages 74–81, 1993.

[26] A. Lakhotia. Understanding someone else's code: Analysis of experience. *Journal of Systems and Software*, 23:269–275, 1993.

[27] S.P. Reiss. Pecan: Program development systems that support multiple views. *IEEE Transactions on Software Engineering*, SE-11(3):276–285, March 1985.

[28] H.A. Müller and K. Klashinsky. Rigi — A system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering (ICSE '10),* (Raffles City, Singapore; April 11-15, 1988), pages 80–86, April 1988.

[29] V. Rajlich, N. Damaskinos, and P. Linos. VIFOR: A tool for software maintenance. *Software–Practice and Experience*, 20(1):67–77, January 1990.

[30] M. Steckel K. Brade, M. Guzdial and E. Soloway. Whorf: A visualization tool for software maintenance. In *Proceedings 1992 IEEE Workshop on Visual Languages,* (Seattle, Washington: Sept 15-18,1992), pages 148–154, 1992.

[31] P. Linos, P. Aubet, L. Dumas, Y. Helleboid, P. Lejeune, and P. Tulula. Visualizing program dependencies: An experimental study. *Software–Practice and Experience*, 24(4):387–403, April 1994.

[32] A. Mendelzon and J. Sametinger. Reverse engineering by visualizing and querying. *Software – Concepts and Tools*, 16:170–182, 1995.

[33] Imagix 4D. Imagix Corporation. http://www.imagix.com/index.html.

[34] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[35] T. Ball and S.G. Eick. Visualizing program slices. In *Proceedings 1994 IEEE Symposium on Visual Languages*, pages 288–295, 1994.

[36] P.E. Livadas and S.D. Alden. A toolset for program understanding. Technical report, University of Florida, 1994.

[37] M.J. Baker and S.G. Eick. Software visualization in the large. *IEEE Computer*, pages 33–43, April, 1996.

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

[39] D. Kimelman, B. Leban, T. Roth, and D. Zernik. Dynamic graph abstraction for effective software visualization. *The Australian Computer Journal*, 27(4):129–137, November 1995.

[40] S.G. Eick and G.J. Wills. Navigating large networks with hierarchies. In *Proceedings Visualization '93,* (San Jose, California: October 25-29,1993), pages 204–210, October 1993.

[41] P. Linos, P. Aubet, L. Dumas, Y. Helleboid, P. Lejeune, and P. Tulula. Facilitating the comprehension of C programs: An experimental study. In *Workshop on Program Comprehension (WPC'93)*, pages 55–63, 1993.

[42] Y.-F. Chen, M.Y. Nishimoto, and C.V. Ramamoorthy. The C information abstraction system. *IEEE Transactions on Software Engineering*, 16(1):325–334, March 1990.

[43] S.R. Tilley, K. Wong, M.-A.D. Storey, and H.A. Müller. Programmable reverse engineering. *International Journal of Software Engineering and Knowledge Engineering*, 4(4):501–520, December 1994.

[44] K. Wong, S.R. Tilley, H.A. Müller, and M.-A.D. Storey. Structural redocumentation: A case study. *IEEE Software*, 12(1):46–54, January 1995.

[45] H.A. Müller, S.R. Tilley, M.A. Orgun, B.D. Corrie, and N.H. Madhavji. A reverse engineering environment based on spatial and visual software interconnection models. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments (SIGSOFT '92),* (Tyson's Corner, Virginia; December 9-11, 1992), pages 88–98, December 1992. In *ACM Software Engineering Notes*, 17(5).

[46] V. Rajlich, J. Doran, and R.T.S. Gudla. Layered explanations of software: A methodology for program comprehension. In *Workshop on Program Comprehension,* Washington, D.C., pages 46–52, November 1994.

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

[48] M.-A.D. Storey, H.A. Müller, and K. Wong. Manipulating and documenting software structures. In P. Eades and K. Zhang, editors, *Software Visualization*, pages 244–263. World Scientific Publishing Co., Fall 1996.

[49] D.A. Penny. *The Software Landscape: A Visual Formalism for Programming-in-the-Large.* PhD thesis, Department of Computer Science, University of Toronto, 1992.

[50] M.P. Consens, F.Ch. Eigler, M.Z. Hasan, A.O. Mendelzon, E.G. Noik, A.G. Ryman, and D. Vista. Architecture and applications of the Hy+ system. *IBM Systems Journal*, 33(4):458–476, August 1994.

[51] M. Heinrichs. Continuous zoom: A java-based graphical user interface for navigating hierarchical structures. http://fas.sfu.ca/ heinrica/personal/CZoom/, October, 1997.

[52] P. Oman. Maintenance tools. *IEEE Software*, pages 59–65, May 1990.

[53] W. Stacy and J. MacMillian. Cognitive bias in software engineering. *Communications of the ACM*, 38(6):57–63, June 1995.

[54] S.P. Reiss. *The Field Programming Environment: A Friendly Integrated Environment for Learning and Development.* The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1995.

[55] C. Rich and R.C. Waters. A research overview. *IEEE Computer*, pages 11–25, November 1988.

[56] H. Koike. The role of another spatial dimension in software visualization. *ACM Transactions on Information Systems*, 11(3):266–286, July 1993.

[57] S.P. Reiss. An engine for the 3d visualization of program information. *Journal of Visual Languages and Computing*, 6:299–323, 1995.

[58] E.L. Burd, P.S. Chan, I.M.M. Duncan, M. Munro, and P. Young. Improving visual representations of code. Technical Report 10/96, University of Durham, Centre for Software Maintenance, October, 1992.

[59] E.R. Tufte. *Envisioning Information*. Graphics Press, 1990.

[60] C. Plaisant, D. Carr, and B. Shneiderman. Image browsers: Taxonomy, guidelines, and informal specifications. Technical report, Human Computer Interaction Laboratory, University of Maryland, August 9, 1994.

[61] Y.K. Leung and M.D. Apperley. A review and taxonomy of distortion-oriented presentation techniques. *ACM Transactions on Computer-Human Interaction*, 1(2):126–160, June 1994.

[62] C. Ware, D. Hui, and G. Franck. Visualizing object oriented software in three dimensions. In *CASCON'93 Proceedings*, 1993.

[63] W. Citrin, C. Santiago, and B. Zorn. Scalable interfaces to support program comprehension. In *4th Workshop on Program Comprehension (WPC'96),* Berlin, Germany, pages 123–132, March, 1996.

[64] D. Flanagan. *Java Tutorial*. O'Reilly, February 1996.

[65] D. A. Norman. *The Design of Everyday Things*. Currency and Doubleday, 1990.

[66] M.J. Baker and S.G. Eick. Space-filling software visualization. *Journal of Visual Languages and Computing*, 6:119–133, 1995.

[67] M.-A.D. Storey, K. Wong, F.D. Fracchia, and H.A. Müller. On integrating visualization techniques for software exploration. In *IEEE Symposium on Information Visualization (INFOVIS '97),* Phoenix, Arizona., pages 38–45, October, 1997.

[68] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[69] B.B. Bederson and J.D. Hollan. Pad++: A zooming graphical interface for exploring alternate interface physics. In *User Interface Software Technology, 1994*, pages 17–26, November 2-4, 1994.

[70] SNiFF+ 2.3. User's Guide and Reference, TakeFive Software. http://www.takefive.com, December, 1996.

[71] B. Bellay and H. Gall. A comparison of four reverse engineering tools. In *Proceedings of the 4th Working Conference on Reverse Engineering (WCRE'97),* Amsterdam, The Netherlands, pages 2–11, October, 1997.

[72] R. Holt and J.Y. Pak. Gase: Visualizing software evolution-in-the-large. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE'96),* Monterey, California, pages 163–167, November, 1996.