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

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**, *provide graphical representations of 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.*

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. The motivation to understand existing programs is obvious; how to design a tool to help the comprehension process is not so obvious.

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

Our long term goal is to design *software exploration tools* which combine graphical representations of software structures linked to textual representations of source code and documentation as an aid in program comprehension. Software visualizations are similar in style to *hypermedia documents*. A hypermedia document contains related and linked representations of an information space [1]. Many of the difficulties experienced by a hyperdocument reader are the same difficulties as those experienced by the browser of a software visualization.

In this paper, we develop a hierarchy of cognitive design elements to be considered when designing software exploration tools. The first branch of this hierarchy addresses issues identified through examination of the cognitive theories of program comprehension and the second branch addresses issues which may reduce a maintainer's cognitive overhead when browsing and navigating large software structures and documentation.

The remainder of this paper is organized as follows. Section 2 describes several cognitive theories of program comprehension. Section 3 describes the various classes of tools which may aid in program comprehension. Section 4 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 given. Section 5 discusses how the hierarchy of design elements may be applied to the design and evaluation of software exploration tools. Section 6 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. All of these cognitive models use existing knowledge together with the code and documentation to create a mental representation of the program [2].

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 [3].

Shneiderman and Mayer's cognitive framework differentiates between syntactic and semantic knowledge of programs [4]. 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 [5] also has a bottom-up flavour. 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 [6] 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* [6]. A beacon is a set of features that indicates the existence of hypothesized structures or operations. An example of a beacon may be a function called `swap` in a sorting program. The discovery of a beacon permits code features to be *bound* to hypotheses.

Soloway and Ehrlich [7] 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 [7]:

- *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 decompose goals and plans into lower level plans.

2.3 Knowledge-based Understanding Model

Letovsky [8] 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 the mental model consists of a specification of the program goals. It later evolves into a mental model which describes the implementation in terms of the data structures and algorithms used. Finally the mental model includes a mapping from the specified program goals to the relevant parts of the implementation.

- The *assimilation process* describes how the mental model evolves using the programmer's knowledge base and 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. *Inquiry episodes* are the central activity in the assimilation process. Inquiry episodes consist of a programmer asking a question (for example, what is the purpose of variable x), conjecturing an answer (x stores the maximum of a set of numbers), and then searching through the code and documentation to verify the answer (the conjecture might be verified if x is in an assignment statement where two values are compared to see which is greater).

2.4 Systematic and As-Needed Program Understanding Strategies

Littman *et al.* [9] 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 only on the code related to a particular task at hand. Soloway *et al.* [10] describe a model which merges the concepts of systematic strategies, as-needed strategies and inquiry episodes (as defined by Letovsky) 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. There are two 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. However it is the most commonly used strategy.

2.5 An Integrated Metamodel of Program Comprehension

Von Mayrhauser and Vans' [2] metamodel integrates Soloway's top-down model with Pennington's program and situation models. From their experiments they observed some programmers frequently switching between all three comprehension models [2]. 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 abstractions 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 [11]. In the integrated model, a situation model may be developed after a partial program model has been formed using systematic or opportunistic understanding strategies [12].
- The *knowledge base* consists of the information needed to build these three cognitive models. It refers to initial knowledge that the programmer has before the maintenance task and is used to store new and inferred knowledge.

Understanding is built at several levels of abstraction simultaneously by switching between the three comprehension processes [12]. According to this model any of the three comprehension processes may be activated at any time [2]. 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 affect the comprehension process must be considered.

Most researchers acknowledge that certain factors will influence the comprehension strategy adopted by a programmer. Vessey [13] states that we must control the factors which influence programmer performance. He specifically mentions program layout, language design, programming mode and programming support facilities. Brooks [6] 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 [14] discriminate between the different strategies required for programs of varying sizes and different tasks. Tilley *et al.* [15] describe how the experience and creativity

Table 1: Influences on Program Comprehension Strategies

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

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. Many researchers strive to limit factors which could influence their experiments, but with the affect that their results are then dependent on the controlled factors. Even if these characteristics could be controlled in a laboratory to perform an experiment, in the real-world these factors cannot be tampered with. If a tool is to aid in comprehension, it must help a maintainer in the key activities identified by the cognition model which bests suits the given characteristics of the maintainer, program and task. However, 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 briefly discusses how a wide variety of tools are used to support program comprehension.

3 Program Comprehension Tools

Understanding programs is often difficult because the source code may be the only source of information. *Reverse engineering* describes the extraction of high-level design information from source code. It is part of the maintenance process that helps a maintainer understand a program so that the program may be altered in some way.

Reverse engineering is done to identify a system's components and their inter-relationships and creates representations of the system in another form, usually at a higher level of abstraction [16]. These higher levels of abstraction are generally less implementation-dependent and more application dependent. Often this information is presented graphically. It is generally accepted that graphical representations are useful as comprehension aids, but "creating and maintaining them continues to be a bottleneck in the process" [16].

Tilley *et al.* [15] describe a conceptual framework for the classification of reverse engineering tools that aid in program comprehension. To develop this classification, they identified three basic activity sets that are characteristic of the reverse engineering process:

- *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.

The exploration of information is the most important of these activities since it "holds the key to program understanding" [15]. The scope of software visualization tools which exist for the navigation, analysis and presentation of software information varies widely [17]. Several software visualization tools show animations of algorithms and data structures. These tools are frequently used in educational settings 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 to understand run-time behaviour. Other tools focus on showing textual representations, some of which may be *pretty printed* to increase understanding.

This paper is concerned with the class of software visualization tools designed for exploring software structure. The next section describes a hierarchy of cognitive issues which should be considered when designing a tool to assist in the exploration of software structures.

4 Cognitive Design Elements for Software Exploration Tools

Software exploration tools typically 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.

Software exploration tools are similar in flavour 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.* [1] 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 from reading the document. 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 [1]. Using the program comprehension models outlined in Section 2, we develop a related hierarchy to guide the development of a tool to aid in the exploration and comprehension of software systems (see Fig. 1). 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 other branch addresses the cognitive issues of the maintainer while he or she browses and navigates the visualization of the software structure. This second branch is similar to those issues which are also relevant for readers of hyperdocuments.

4.1 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. Although often it is preferable to develop specialized tools which suit a particular comprehension strategy. 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.

4.1.1 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 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 obvious or 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 [18], Rigi [19], VIFOR [20], Whorf [21], CARE [22], Hy+ [23], Imagix 4D [24] 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 which will quickly lead to a feeling of disorientation.

Whorf [21] 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 colour 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 behaviour [25]. SeeSlice [26] is a tool for visualizing program slices where program files are displayed

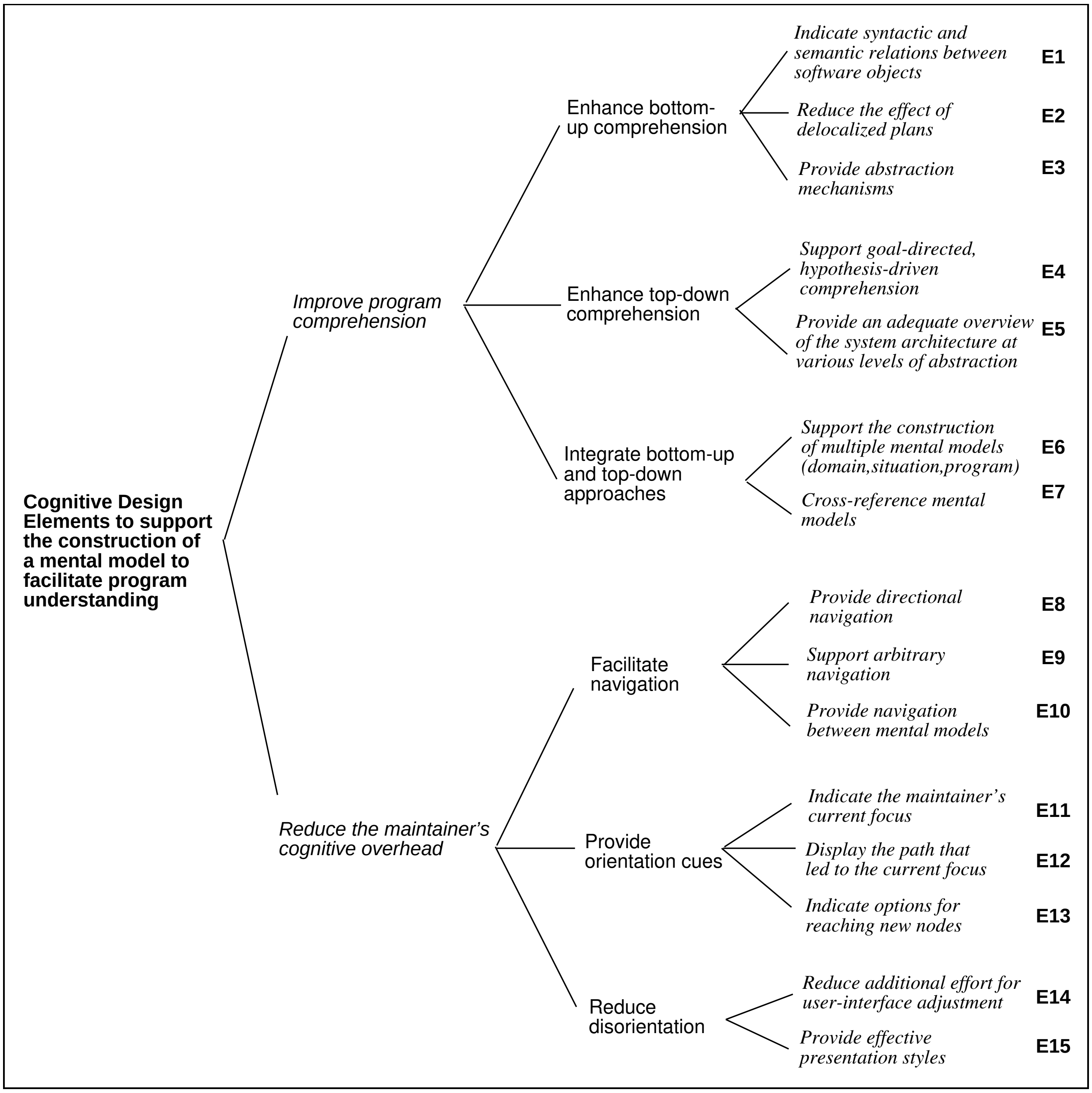

Figure 1: Cognitive Design Elements for Software Exploration

as columns that contain line representations of procedures. Procedures may be displayed as *open*, where each line of code is displayed as a thin row indented accordingly. Code that is not part of a slice is elided. Ghinsu [27] (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 decomposition process (the process of building hierarchical abstractions from the low level software objects and relations) is the hardest part of bottom-up comprehension, and yet many tools only support showing a previously decomposed view [28]. 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 tools, a subgraph (a set of nodes and arcs) may be *collapsed* into a single *composite* or *subsystem* node [29, 30, 24, 31, 22, 32]. Several tools also provide the ability to filter (temporarily elide) objects which results in a less detailed or abstract graph [26, 29, 30, 24, 22, 32, 20].

4.1.2 Enhance top-down comprehension

Understanding a program top-down requires application domain knowledge or previous exposure to the program. 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. 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 [33]. 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, useful documentation for future maintenance [11]. However, TLES is more suited to documenting evolving systems rather than as a redocumentation tool.

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 decomposes 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 each contain a slice of the hierarchy at selected levels of abstraction [34].

Landscape views [35], Hy+ [36] and SHriMP views [34] use a *nested graph* representation of the 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.

4.1.3 Integrate bottom-up and top-down approaches

Von Mayrhauser and Vans observed programmers using both bottom-up and top-down approaches [11]. Programmers create various mental models and 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 the graph model. Displaying how functional abstractions relate to the application domain is a harder task. Another model which can be visually presented shows the behaviour 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 [37]. 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 (PLUM [18], Rigi [38], Whorf [21], Garden

[39], VIFOR [20], CARE [22], SeeSlice [26], and Imagix [24]). Example graphical views show call-graphs and variable usage diagrams. Example textual views include displaying source code, program slices and software statistics (metrics). Some tools, such as PECAN [18], also support a view showing the execution of the program. Multiple views are often shown side by side, or displayed using overlapping and 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 [11].

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

Maintainers frequently switch one from one model to another in the course of comprehension [12]. 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 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 Rigi [38], PECAN [18], Whorf [21], CARE [22] and Imagix [24]. 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 [23].

The Programmer's Apprentice tool has direct support for both development of the program model and situation model [40]. This tool uses *Plan Calculus* to formally represent programs and clichés (plans) [40]. Plan Calculus has a graphical notation and a formal semantics which can be used to show a mapping between an abstraction of the 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 show correspondences between the two diagrams.

4.2 Reduce the Maintainer's Cognitive Overhead

When comprehending larger software systems, the cognitive overhead increases rapidly. Visualization tools are often supplied in an effort to reduce this 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 [1].

Although a tool may provide many navigation methods and effective orientation cues, the user may still feel overwhelmed if they are presented with too much information. Effective presentation techniques can alleviate the effects of displaying large amounts of information [41]. Disorientation may also result from a badly designed user interface which lacks in a feeling of continuity between displays [1].

4.2.1 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 it may be browsed 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 with hypertext links. In Imagix [24], code and documentation is generated in HTML format to be browsed by a web browser. Subsystem hierarchies are navigated in Rigi [38], Whorf [21], CARE [22] and Imagix [24] by selecting subsystem nodes to open a window to show a view of the subsystem node selected.

E9: ***Support arbitrary navigation***

Arbitrary navigation is supported when a maintainer may navigate to locations not necessarily reachable by following an application or user-defined link. 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 [18] and Rigi [38]) may be used as a mechanism for storing arbitrary navigation steps. A maintainer creates a snapshot of the current view so that it may be accessed in the future without having to follow defined links in the software visualization. Searching capabilities are available in several tools to provide another mechanism for navigation [24, 21, 22, 19].

E10: ***Provide navigation between mental models***

Navigation between the various mental models is the key to successfully using them for comprehension [11]. 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 implicitly 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.

4.2.2 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.

E11: ***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.

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. Indicating the maintainer's current focus, requires not only showing the artifacts that are of immediate interest, but also displaying context for those artifacts. Many systems, such as Rigi [38] and Whorf [21], use highlighted nodes and arcs to emphasize the current focus in a graph. In larger graphs highlighted nodes and arcs may not always be obvious. Some software visualization systems (Hy+ [23], PLUM [42] and SHriMP Views [34]) make use of *fisheye* display techniques [43] by allocating more screen space to more important information by displaying it larger than secondary information.

E12: ***Show the path that led to the current focus***

In hypermedia document browsers there are often *histories* of traveled paths to indicate to the reader how a particular document in the structure was reached. Similar facilities may be used for browsing software documents. 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 may be very important. 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.

E13: ***Indicate the options for reaching new nodes***

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 [24, 44]. 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 [45] 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.

4.2.3 Reduce disorientation

For the exploration of larger systems, reducing disorientation effects is critical. 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.

E14: ***Reduce additional effort for user-interface adjustment***

Poorly designed interfaces will of course induce extra overhead. Available functionality should be visible and relevant [46] 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 [47] 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 [31]. Although there is extra overhead involved in computing the transitions between views, the effects of reduced disorientation may be worth the effort.

E15: ***Provide effective presentation styles***

For complex graphs typical of larger software systems, layout algorithms are used to display the graph in a more meaningful manner. Although software has no inherent shape or colour, 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 will give the impression of increased complexity in the software.

Many software visualization tools (VIFOR [20], CIA [29], CARE [22], Hy+ [36], HierNet [30], PLUM [42], SeeSys [47] and Rigi [48]) recognize the importance of graph layouts and provide specialized or customized layouts suitable for presenting software graphs.

5 Towards an Effective Interface for Software Exploration

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 is also an area for future research.

We have begun to apply the cognitive design elements developed in Section 4 to the design of a software visualization technique called SHriMP (**S**imple **Hi**e**r**archi**c**al **M**ulti-**P**erspective) views [49]. This approach has been integrated in the Rigi system and was developed in response to some deficiencies identified with the original visualization methods used in Rigi [49]. The original interface consisted of a multiple window approach for displaying software structures. It was observed that some users frequently lost context due to the lack of orientation cues (E11,E12,E13). In addition, users had difficulties switching from one view to another when new windows were opened (E15).

The SHriMP technique uses a fisheye view of nested graphs to show a single view of the software structure. The nested graph display of software structures is useful for displaying multiple levels of abstraction (E5) and provides effective orientation cues lacking in the original interface (E11,E12,E13). The interactive fisheye view should reduce cognitive overhead for user interface adjustment since it shows both context and detail in a single view (E15). It is hoped that the SHriMP approach will address issues relevant to the integrated model of program comprehension by supporting switching between mental models at various levels of abstraction (E7).

A preliminary experiment was performed to evaluate the effectiveness of this new interface in Rigi [50]. Although the focus of this initial experiment was to refine the experiment design, the results were encouraging. The experiment methodology developed for this initial test will be used to conduct larger scale experiments in the near future. We intend to use the hierarchy of design elements as a blueprint against which the distinctive features of the SHriMP and Rigi interfaces may be compared from a cognitive perspective. The results and observations from these experiments will drive design decisions in subsequent reimplementations of the SHriMP interface. Using this iterative cycle of design and test, we are working towards a more effective interface for software exploration.

6 Conclusions

This paper described a hierarchy of cognitive issues which should be considered during the design of software exploration tools. The hierarchy of issues was derived through examination of the cognitive models of program comprehension. Examples of how existing software exploration tools address each of these issues was provided.

In [15], Tilley *et al.* describe a framework for classifying reverse engineering tools. The hierarchy of design elements developed in this paper can be used for a more detailed classification of tools which are geared towards the analysis, presentation and navigation of software information. In this paper, we illustrated how several software visualization tools address each of the cognitive design elements. An extension of this work would be to use the hierarchy of design elements as a basis for a taxonomy to compare and contrast software exploration tools.

In general, there has been a lack of evaluation of software exploration tools. Hopefully, this trend will not continue. The hierarchy of cognitive design elements identified in this paper may be used for selecting criteria to evaluate software exploration tools.

The hierarchy of cognitive design elements raises our awareness of the issues and difficulties which should be addressed in the design of software exploration tools. Although the hierarchy in itself does not provide enough insight as to how these issues should be resolved, it may be used for deriving sets of guidelines when making key interface design decisions.

References

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

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

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

[4] 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.

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

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

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

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

[9] 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.

[10] 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.

[11] 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, 19-23 July, 1993.

[12] 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.

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

[14] 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.

[15] S.R. Tilley, S. Paul, and D.B. Smith. Towards a framework for program understanding. In *WPC'96: 4th Workshop on Program Comprehension,* Berlin, Germany, March 29-31, 1996.

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

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

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

[19] 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. IEEE Computer Society Press (Order Number 849).

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

[21] 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.

[22] 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.

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

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

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

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

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

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

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

[36] 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*, August 1994.

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

[38] H.A. Müller, B.D. Corrie, and S.R. Tilley. Spatial and visual representations of software structures: A model for reverse engineering. Technical Report TR-74.086, IBM Canada Ltd., April 25, 1992.

[39] 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.

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

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

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

[43] 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.

[44] David Flanagan. *Java Tutorial*. O'Reilly, February 1996.

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

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

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

[48] 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), December 1994.

[49] 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.

[50] 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 (WCRE96),* Monterey, California, Nov 8-10, 1996.