Theories, Methods and Tools in Program Comprehension: *Past, Present and Future*

Margaret-Anne Storey
Department of Computer Science,
University of Victoria, Canada
E-mail: mstorey@uvic.ca

Abstract

Program comprehension research can be characterized by both the theories that provide rich explanations about how programmers comprehend software, as well as the tools that are used to assist in comprehension tasks. During this talk I will review some of the key cognitive theories of program comprehension that have emerged over the past thirty years. Using these theories as a canvas, I will then explore how tools that are popular today have evolved to support program comprehension. Specifically, I will discuss how the theories and tools are related and reflect on the research methods that were used to construct the theories and evaluate the tools. The reviewed theories and tools will be further differentiated according to human characteristics, program characteristics, and the context for the various comprehension tasks. Finally, I will predict how these characteristics will change in the future and speculate on how a number of important research directions could lead to improvements in program comprehension tools and methods.

1. Introduction

Challenges in understanding programs are all too familiar from even before the days of the first software engineering workshop [1]. Since that time, the field of program comprehension as a research discipline has evolved considerably. The goal of our community is to build an understanding of these challenges, with the ultimate objective of developing more effective tools and methods. From these early days we have come to accept that there is no silver bullet [2], but the community has made advances which have helped software engineers tackle important problems such as the Y2K problem.

We now have a wide variety of theories that provide rich explanations of how programmers understand programs and can provide advice on how program comprehension tools and methods may be improved. In response to these theories, and in some cases in parallel to the theory development, many powerful tools and innovative software processes have evolved to improve comprehension activities.

The field of program comprehension research has been rich and varied, with various shifts in paradigms and research cultures during the last few decades. A multitude of differences in program characteristics, programmer ability and software tasks have led to many diverse theories, research methods and tools. In this paper, I provide a review of this work in an attempt to create a landscape of program comprehension research. Such a view emphasizes how the theories and tools are related and should reveal if parts of the landscape have not received much attention. This review, combined with an excursion to newer areas of software engineering theory and practice, directs us to specific areas for the future of program comprehension research.

This paper is organized as follows. In Section 2, I provide a brief overview of comprehension theories and describe how programmer, program and task variability can impact comprehension strategies. In Section 3, the implications of cognitive theories on tool requirements are considered and several theories that specifically address tool support are reviewed. In Section 4, I briefly describe comprehension tools and refer back to the theories about tool support. In Section 5, I look to the future and predict how programmer and program characteristics are likely to vary in the near term. Building on these predicted changes, I then suggest, in Section 6, how research methods, theories and tools will evolve in the future. The paper concludes in Section 7.

2. A review of cognitive theories

Francois Détienne's book, "Software Design - Cognitive Aspects" [3], provides an excellent review of the history of cognitive models and related experiments over the past twenty or so years. She delves back to a time, in the early 1970's, when experiments were done without theoretical frameworks to guide the evaluations. Consequently, it was neither possible to understand nor to explain to others why one tool might be superior to other tools.

The lack of theories was recognized as being problematic. As the field of program comprehension matured, research methods and theories were borrowed from other areas of research, such as text comprehension, problem solving and education. Using these theoretical underpinnings, cognitive theories about how programmers understand programs and how tools could support comprehension were developed. Perceived benefits of having these models include having rich explanations of behaviour that would lead to more efficient processes and methods as well as improved education procedures [4].

In this section, I first review some of the influential cognitive theories in program comprehension research. I then discuss how the programmer, program and task characteristics impact comprehension. First, some terminology is defined.

2.1 Concepts and terminology

A ***mental model*** describes a developer's mental representation of the program to be understood whereas a ***cognitive model*** describes the cognitive processes and temporary information structures in the programmer's head that are used to form the mental model. ***Cognitive support*** assists cognitive tasks such as thinking or reasoning [5].

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 [6].

Beacons are recognizable, familiar features in the code that act as cues to the presence of certain structures [7]. ***Rules of programming*** discourse capture the conventions of programming, such as coding standards and algorithm implementations [6].

2.2 Top-down comprehension

Brooks 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 this knowledge to the source code [7]. The process starts with a hypothesis about the general nature of the program. This initial hypothesis is then refined in a hierarchical fashion by forming subsidiary hypotheses. Subsidiary hypotheses are refined and evaluated in a depth-first manner. The verification (or rejection) of hypotheses depends heavily on the absence or presence of beacons [7].

Soloway and Ehrlich [6] observed that top-down understanding is used when the code or type of code is familiar. They observed that expert programmers use beacons, programming plans and rules of programming discourse to decompose goals and plans into lower-level plans. They noted that *delocalized plans* complicate program comprehension.

2.3 Bottom-up comprehension

The bottom-up theory of program comprehension assumes that programmers first read code statements and then mentally chunk or group these statements into higher level abstractions. These abstractions (chunks) are aggregated further until a high-level understanding of the program is attained [8]. Shneiderman and Mayer's cognitive framework differentiates between *syntactic* and *semantic* knowledge of programs [8]. 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.

Pennington also describes a bottom-up model [9]. 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) 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 and functional abstractions (the program goal hierarchy).

2.4 Opportunistic and systematic strategies

Littman *et al.* observed programmers enhancing a personnel database program [10]. They observed that programmers either 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 that they take an as-needed approach, focusing only on the code relating to a particular task at hand. Subjects using a systematic strategy acquired both static knowledge (information about the structure of the program) and causal knowledge (interactions between components in the program when it is executed). This enabled them to form a mental model of the program. However, those using the as-needed approach only acquired static knowledge resulting in a weaker mental model of how the program worked. More errors occurred since the programmers failed to recognize causal interactions between components in the program.

Letovsky observed activities called *inquiries* [11]. An inquiry may consist of a programmer asking a question, conjecturing an answer, and then searching through the code and documentation to verify the answer. Inquiry episodes often occur as a result of *delocalized plans*.

2.5 The Integrated Metamodel

The Integrated Metamodel, developed by von Mayrhauser and Vans, builds on the previous three models as well as the knowledge based model by Letovsky [11]. Their model consists of four major components [12]. 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 and developed using an as-needed strategy, when the programming language or code is familiar. It incorporates domain knowledge as a starting point for formulating hypotheses.
- The program model may be invoked when the code and application is completely unfamiliar. The program model is a control-flow abstraction.
- The situation model describes data-flow and functional abstractions in the program. It may be developed after a partial program model is formed using systematic or opportunistic strategies.
- The knowledge base consists of information needed to build these three cognitive models. It represents the programmer's current knowledge 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.

2.6 Program characteristics

Programs that are carefully designed and well documented will be easier to understand, change or reuse in the future. Pennington's experiments showed that the choice of language has an effect on comprehension processes [9]. COBOL programmers consistently fared better at answering questions related to data-flow than FORTRAN programmers, and FORTRAN programmers consistently fared better than COBOL programmers for control-flow questions. Object-oriented (OO) programs are often seen as a more natural fit to problems in the real world because of 'is-a' and 'is-part-of' relationships in a class hierarchy and structure, but others argue that objects do not always map easily to real world problems [3]. In OO programs, abstractions are achieved through encapsulation and polymorphism [3]. Message-passing is used for communication between class methods and hence programming plans are dispersed (i.e. scattered) throughout classes in an OO program.

2.7 Individual programmer differences

There are many individual characteristics that will impact how a programmer tackles a comprehension task. These differences also impact the requirements for a supporting tool. There is a huge disparity in programmer ability and creativity which cannot be measured simply by their experience.

Vessey presents an exploratory study to investigate expert and novice debugging processes [13]. She classified programmers as expert or novice based on their ability to chunk effectively. She notes that experts used breadth-first approaches and at the same time were able to adopt a system view of the problem area, whereas novices used breadth-first and depth-first approaches but were unable to think in system terms. Détienne also notes that experts make more use of external devices as memory aids [3]. Experts tend to reason about programs according to both functional and object-oriented relationships and consider the algorithm, whereas novices tend to focus on objects.

2.8 Task variability

Program comprehension is not an end goal, but rather a necessary step in achieving some other objective, such as fixing an error, reusing code, or making changes to a program. Obviously the type and scope of the ultimate programming task will have an impact on the comprehension process followed. If a task is a simple one, the change will probably only affect a small portion of the code. For more complex changes, the programmer will have to consider global interactions thus requiring that the programmer obtain a thorough understanding of the causal relationships in the entire program.

Pennington's research showed that a task requiring recall and comprehension resulted in a programmer forming a program model (control-flow abstraction) of the software whereas a task to modify the program resulted in a programmer forming a situation model containing data-flow and functional information [9].

For a programmer, a reuse task requires that they first understand the source problem, retrieve an appropriate target solution, and then adapt the solution to the problem. The mapping from the problem to the solution is often done using analogical reasoning [3] and may involve iterative searching through many possible solutions.

2.9 Discussion: *Research implications*

Many of the researchers that developed the traditional cognitive theories for program comprehension discuss the implications of the developed theories on tool design and in some cases also discuss how education and program design could be improved to address program understanding challenges. In many cases, the connection to tools and how they could be improved or evaluated according to the theories could be stronger. Moreover, some of these results were also criticized because the researchers studied novice programmers doing fabricated tasks [14]. Vans and von Mayrhauser are notable exceptions. Despite these criticisms, program comprehension research does contain many gems of advice on how tools can be improved. The advantage for tool designers is that they can use these theories to help them understand, not only what features are needed, but also to help them understand why some features may not be appropriate or sufficient to assist comprehension tasks.

3. Theories and tool support

What features should an ideal tool to support program comprehension have? Of course, "program comprehension tools" only play a supporting role in other software engineering activities of design, development, maintenance, and redocumentation. As we saw in the last section of this paper, there are many characteristics which influence the cognitive strategies the programmers used and in turn influence the requirements for tool support. In this section, I look in general terms at tool requirements but these should be refined for a tool to be deployed in a particular context. First I extract and synthesize tool requirements based on our prior discussion on cognitive theories. I then look at specific research that recommends tool features.

3.1 Cognitive models and tool implications

Browsing support*:* The top-down process requires browsing from high-level abstractions or concepts to lower level details, taking advantage of beacons in the code; bottom-up comprehension requires following control-flow and data-flow links, both novices and experts can benefit from tools that support breadth-first and depth-first browsing; and the Integrated Metamodel suggests that switching between top-down and bottom-up browsing should be supported. Flexible browsing support also will help to offset the challenges from *delocalized plans.*

Searching: Tool support is needed when looking for code snippets by analogy and for iterative searching. Also inquiry episodes should be supported by allowing the programmer to query on the role of a variable, function etc.

Multiple views: Programming environments should provide different ways of visualizing programs. One view could show the message call graph providing insight into the programming plans, while another view could show a representation of the classes and relationships between them to show an object-centric or data-centric view of the program. These orthogonal views, if easily accessible, can facilitate comprehension, especially when combined.

Context-driven views: The size of the program and other program metrics will influence which view is the preferred one to show a programmer browsing the code for the first time. For example, in an object-oriented program, it is usually preferable to show the inheritance hierarchy as the initial view. However, if the inheritance hierarchy is flat, it may be more appropriate to show a call graph as the default view.

Additional cognitive support: Experts need external devices and scratchpads to support their cognitive tasks, whereas novices need pedagogical support to help them access information about the programming language and the corresponding domain.

3.2 Tool requirements explicitly identified

Several researchers studied expert programmers in industrial settings and consequently recommended specific requirements for improving tools to support comprehension. Others built on their own personal or colleagues' experiences to recommend needed tool features. I review several of these efforts here and list the tool requirements they recommended.

Biggerstaff notes that one of the main difficulties in understanding comes from mapping what is in the code to the software requirements – he terms this the concept assignment problem [15]. Although automated techniques can help locate programming concepts and features, it is challenging to automatically detect human oriented concepts. The user may need to indicate a starting point and then use slicing techniques to find related code. It may also be possible for an intelligent agent (that has domain knowledge) to scan the code and search for candidate starting points. From Biggerstaff's research prototypes he found that queries, graphical views and hypertext were important tool features.

Von Mayrhauser and Vans, from their research on the Integrated Metamodel, make an explicit recommendation for tool support for reverse engineering [12]. They determined basic information needs according to cognitive tasks and suggested the following tool capabilities to meet those needs:

- Top-down model: on-line documents with keyword search across documents; pruning of call tree based on specific categories; smart differencing features; history of browsed locations; and entity fan-in.
- Situation model: provide a complete list of domain sources including non-code related sources; and visual representation of major domain functions.
- Program model: Pop-up declarations; on-line cross-reference reports and function count.

K. Wong also discusses reverse engineering tool features [49]. He specifically mentions the benefits of using a "notebook" to support ongoing comprehension.

Singer and Lethbridge also observed the work practices of software engineers [16]. They explored the activities of a single engineer, a group of engineers, and considered company-wide tool usage statistics. Their study led to the requirements for a tool that was implemented and successfully adopted by the company. Specifically they suggested tool features to support "just-in-time comprehension of source code". They noted that engineers after working on a specific part of the program quickly forget details when they move to a new location. This forces them to rediscover information at a later time. They suggest that tools need the following features to support rediscovery:

- Search capabilities so that the user can search for code artifacts by name or by pattern matching.
- Capabilities to display all relevant attributes of the items retrieved as well as relationships among items.
- Features to keep track of searches and problem-solving sessions, to support the navigation of a persistent history.

Erdö**s and Sneed** designed a tool to support maintenance following many years of experience in the maintenance and reengineering industry. They proposed that the following seven questions need to be answered for a programmer to maintain a program that is only partially understood [17]:

1. Where is a particular subroutine/procedure invoked?
2. What are the arguments and results of a function?
3. How does control flow reach a particular location?
4. Where is a particular variable set, used or queried?
5. Where is a particular variable declared?
6. Where is a particular data object accessed?
7. What are the inputs and outputs of a module?

3.3 Discussion: *Ways of knowing*

In our quest to discover effective features to support program comprehension, we see that there are many different ways of knowing. On the one hand, we can use the empirical approach which leads to the construction of theories about program comprehension strategies and proposed tools. On the other hand, researchers may use practical experience and intuition to propose what is needed in a tool. Given the variability in comprehension settings, both approaches contribute to answering this complex question.

4. Tool research

The field of program comprehension research has resulted in many diverse tools to assist in program comprehension. Program understanding tools can be roughly categorized according to three categories [18]: extraction, analysis and presentation. Extraction tools include parsers and data gathering tools. Analysis tools do static and dynamic analyses to support activities such as clustering, concept assignment, feature identification, transformations, domain analysis, slicing and metrics calculations. Presentation tools include code editors, browsers, hypertext, and visualizations. Integrated software development and reverse engineering environments will usually have some features from each category. The set of features they support is usually determined by the purpose for the resulting tool or by the focus of the research.

It is possible to examine each of these environments and to recover the motivation for the features they provide by tracing back to the cognitive and tool theories. For example, the well known Rigi system [19] has support for multiple views, cross-referencing and queries to support bottom-up comprehension. The Reflexion tool [20] has support for the top-down approach through hypothesis generation and verification. The Bauhaus tool [21] has features to support clustering (identification of components) and concept analysis. The SHriMP tool [22] provides navigation support for the Integrated Metamodel, i.e. frequent switching between strategies. And the Codecrawler tool [23] uses visualization of metrics to support understanding of an unfamiliar system and to identify bottlenecks and other architectural features.

Theories also play a role in the evaluation of these environments. They can be used as a first step in performing a heuristic evaluation of the environment and in describing what the environment does and does not do. Secondly, they can be used to help guide evaluations and to assist in presenting results.

5. Programmer and program trends

Any paper or talk that attempts to predict the future, always discusses how risky and difficult such an endeavour is. Fortunately, it is less risky to closely examine current trends and predict what will occur in the very near future. Tilley and Smith, in 1996, wrote a thought-provoking paper entitled "Coming Attractions in Program Understanding" [18]. I follow a similar approach to that of Tilley and Smith, and keep the distance to the horizon short by looking at the near future. To guide theory and tool predictions, I first consider how some of the programmer and program characteristics will evolve in the near future.

5.1 Programmer characteristics

Program comprehension everywhere: The need to use computers and software intersects every walk of life. Programming, and hence program comprehension, is no longer a niche activity. Scientists and knowledge workers in many walks of life have to use and customize software to help them do science or other work. In our research alone, we have already worked alongside scientists from the forestry, astronomy and medical science domains that are using and developing sophisticated software without a formal education in computer science. Consequently, there is a need for techniques to assist in non-expert and end-user program comprehension. Fortunately, there is much work in this area (especially at conferences such as Visual Languages and the PPIG group - www.ppig.org), where they investigate how comprehension can be improved through tool support for spreadsheet and other end user applications.

Sophisticated users: Currently, advanced visual interfaces are not often used in development environments. A large concern by many tool designers is that these advanced visual interfaces require complex user interactions. However, tomorrow's programmers will be more familiar with game software and other media that displays information rapidly and requires sophisticated user controls. Consequently, the next generation of users will have more skill at interpreting information presented visually and at manipulating and learning how to use complex controls.

Globally distributed teams: Advances in communication technologies have enabled globally distributed collaborations in software development. Distributed open source development is having an impact on industry. The most notable examples are Linux and Eclipse. Some research has been conducted on studying collaborative processes in open source projects [24-26], but more research is needed to study how distributed collaborations impact comprehension.

5.2 Program characteristics

Distributed applications and web-based applications are becoming more prevalent with technologies such as .NET, J2EE and web services. One programming challenge that is occurring now and is likely to increase, is the combination of different paradigms in distributed applications, e.g. a client side script sends XML to a server application.

Higher levels of abstraction: Visual composition languages for business applications are also on the increase. As the level of abstraction increases, comprehension challenges are shifting from code understanding to more abstract concepts.

Aspect-oriented programming: The introduction of aspects as a construct to manage scattered concerns (delocalized plans) in a program has created much excitement in the software engineering community. Aspects have been shown to be effective for managing many programming concerns, such as logging and security. However, it is not clear how aspects written by others will improve program understanding, especially in the long term. More empirical work is needed to validate the assumed benefits of aspects.

Improved software engineering practices: The more informed processes that are used for developing software today will hopefully lead to software that is easier to comprehend in the future. Component-based software systems are currently being designed using familiar design patterns, and other conventions. Future software may have traceability links to requirements, and improved documentation such as formal program specifications. Also, future software may have autonomic properties, where the software self-heals and adapts as its environment changes – thus in some cases reducing time spent on maintenance.

Diverse sources of information: The program comprehension community, until quite recently, mostly focused on how static and dynamic analyses of source code, in conjunction with documentation, could facilitate program comprehension. Modern software integrated development environments, such as the Eclipse Java development environment, also manage other kinds of information such as bug tracking, test cases and version control. This information, combined with human activity information such as emails and instant messages, will be more readily available to support analysis in program comprehension. Domain information should also be more accessible due to model driven development and the semantic web.

6. Future methods, theories and tools

In this section, I now consider how methods, theories and tools may evolve in response to the predicted programmer and program changes.

6.1 Research methods

In recent years there has been a high expectation in the research community that tools should be subject to some sort of evaluation. Case studies of industrial systems are often used as a mechanism for demonstrating that a technique can efficiently and robustly perform the expected analysis. In some cases, we may also need evidence that the approach is useful for and usable by less sophisticated developers.

Many techniques, especially those utilized in presentation tools, are evaluated using experiments. Conducting empirical work and experiments in program comprehension is always a challenging endeavour. There are two main research paradigms, the ***quantitative*** and the ***qualitative*** [27]. The quantitative approach, the traditional approach in program comprehension, assumes that reality is objective and is independent from the researcher. Quantitative studies are more formal and factors are isolated before the study. They are performed in context free situations, and the studies are seen as reliable and repeatable. The qualitative approach, the constructivist approach, assumes that the researcher interacts with what is observed and that context is important. The study is more informal and hypotheses about the results are formed inductively. The key results from qualitative research are patterns and theories which lead to initial or further understanding.

For the most part, cognitive models and tool evaluations in program comprehension have been determined through quantitative approaches. However, the conditions for program comprehension are so complex and varied, that many researchers are recognizing that qualitative approaches conducted in more ecologically valid settings may be very insightful. This shift to conducting research in industrial settings brings with it many logistical challenges. Observations can be hard to do in industry and may result in vast amounts of data that is difficult to analyze. Observations can also be disruptive and could be subject to the Hawthorne effect (e.g. a programmer may change her behaviour because she is observed).

To address these issues, several researchers are also collecting instrumented data after the tool is deployed in an industrial setting. This data collection technique shows much promise as it captures information on the context of tool use as well as accurate information about how a tool is used over a longer period of time. However, such results can also be misleading. A lack of adoption is not enough to indicate that a tool is not useful as there are many barriers to adoption (e.g. seemingly trivial usability issues can impede usage of a tool). There have been several ICSE workshops (2003, 2004) which have discussed adoption of software tools.

As a community, there have been some shared evaluation efforts -- the use of guinea pigs (i.e. benchmarks) and collaborative tool demonstrations [28]. These efforts give researchers a mechanism to compare their tools with others, and learn more about different and similar approaches to research.

Irrespective of the evaluation technique used, theoretical underpinnings will benefit the evaluations as the results will be easier to interpret. Although our long term goal may be to build better tools, we need to understand *why* they are better than other approaches.

6.2 Theories

As the programming workforce and technology changes, learning theories [29] will become more relevant to end-users doing programming-like tasks. Theories are currently being developed to describe the social and organizational aspects of program comprehension [25]. Richer cognitive theories about how aspect oriented programming will impact comprehension in the longer term need to be further developed. More theories about the collaborative nature of program comprehension, both co-located and distributed, are needed.

It is becoming clear to many in the field, that developing theories on program comprehension is an ambitious undertaking. This is mostly due to the large variability in the possible conditions for any experiment. It is important as a community to have many data points; this will enable future researchers to do a meta-analysis of results from several experiments so that common trends and issues can be extracted. This phenomenon can be compared to efforts in the clinical trial community where many studies have to be done to understand how a drug interacts with other drugs and different kinds of individuals.

In our research community, we need to document and present results in such a way that others can make sense of our data and conclusions. Researchers evaluating presentation tools and user interfaces for program comprehension tools could benefit from the work of Walenstein and Green *et al.:* Walenstein proposes a methodology for evaluating cognitive support in [5] and Green's Cognitive Dimensions [30] provides a language and framework that can be used for evaluating presentation tools. More work is needed to understand how we can combine results and benefit from collaborative efforts in empirical work.

6.3 Tools

Faster tool innovations: The use of frameworks as an underlying technology for software tools is leading to faster tool innovations as less time needs to be spent reinventing the wheel. A prime example of how frameworks can improve tool development is the Eclipse framework (see www.eclipse.org). Eclipse was specifically designed with the goal of creating reusable components which would be shared across different tools. The research community benefits from this approach in several ways. Firstly, they are able to spend more time writing new and innovative features as they can reuse the core underlying features offered by Eclipse and its plug-ins; and secondly, researchers can evaluate their prototypes in more ecologically valid ways as they can compare their new features against existing industrial tools.

Mix 'n match tools: In Section 5, I described understanding tools according to three categories: extraction, analysis and presentation. Given a suite of tools that all plug in to the same framework, together with a standard exchange format (such as GXL), researchers will be able to more easily try different combinations of tools to meet their research needs. This should result in increased collaborations and more relevant research results. Such integrations will also lead to improved accessibility to repositories of information related to the software including code, documentation, analysis results, domain information and human activity information. Integrated tools will also lead to fewer disruptions for programmers.

Recommenders and search: Software engineering tools, especially those developed in research, are increasingly leveraging advances in intelligent user interfaces (e.g. tools with some domain or user knowledge). Recommender systems are being proposed to guide navigation in software spaces. Examples of such systems include Mylar [31] and NavTracks [32]. Mylar uses a degree of interest model to filter non-relevant files from the file explorer and other views in Eclipse. NavTracks provides recommendations of which files are related to the currently selected files. Deline *et al.* also discuss a system to improve navigation [33]. The FEAT tool suggests using concern graphs (explicitly created by the programmer) to improve navigation efficiency and enhance comprehension [34].

Search technologies, such as Google, show much promise at improving search for relevant components, code snippets and related code. The Hipikat tool [35] recommends relevant software artifacts based on the developer's current project context and development history. The Prospector system recommends relevant code snippets [36]. It combines a search engine with the content assist in Eclipse to help programmers use complex APIs. A paper in press [37] also proposes using structures to create code recommendations during evolution tasks.

Although this work in search and recommendations is quite new, it shows much promise and it is expected to improve navigation in large systems while reducing the barriers to reusing components from large libraries.

Adaptive interfaces: Software tools typically have many features which may be overwhelming not only for novice users, but also for expert users. This information overload could be reduced through the use of adaptive interfaces. The idea is that the user interface can be tailored automatically, i.e. will self-adapt, to suit different kinds of users and tasks. Adaptive user interfaces are now common in Windows applications such as Word. Eclipse has several novice views (such as Gild [38] and Penumbra) and Visual Studio has the Express configuration for new users. However, neither of these mainstream tools currently have the ability to adapt or even be easily manually adapted to the continuum of novice to expert users.

Visualizations have been the subject of much research over the past ten to twenty years. Many visualizations, and in particular graph-based visualizations, have been proposed to support comprehension tasks. Some examples of research tools include Seesoft [39], Bloom [40], Rigi [41], Landscape views [42], sv3D [43], and Codecrawler [23]. In our work, we developed the SHriMP tool [22] to provide support for the Integrated Metamodel of comprehension. Our goal was also to facilitate navigation between different layers of abstractions, and to help navigation through delocalized plans. Graph visualization is used in many advanced commercial tools such as Klocwork, Imagix4D and Together. UML diagrams are also common place in mainstream development tools.

One challenge with visualizing software is scale and knowing at what level of abstraction details should be shown, as well as selecting which view to show. More details about the user's task combined with metrics describing the program's characteristics (such as inheritance depth) will improve how visualizations are currently presented to the user. A recommender system could suggest relevant views as a starting point. Bull proposes the notion of ***model driven visualization*** [44]. He suggests creating a tool for tool designers and expert users that recommends useful views based on characteristics of the model and data.

Collaborative support: As software teams increase in size and become more distributed, collaborative tools to support distributed software development activities are more crucial. In research, there are several collaborative software engineering tools being developed such as Jazz and Augur [45, 46]. There are also some collaborative software engineering tools deployed in industry, such as CollabNet, but they tend to have simple tool features to support communication and collaboration, such as version control, email and instant messaging. Current industrial tools lack more advanced collaboration features such as shared editors. Although collaborative tools for software engineering have been a research topic for several years, there has been a lack of adoption of many of the approaches such as shared editors in industry and a lack of empirical work on the benefits of these tools. Another area for research that may prove useful is the use of large screen displays to support co-located comprehension. O'Reilly *et al.* [47] propose a war room command console to share visualizations for team coordination. There are other research ideas in the CSCW (computer supported collaborative work) field that could be applied to program comprehension.

Domain and pedagogical support: The need to support domain experts that lack formal computer science training will necessarily result in more domain-specific languages and tools. Non-experts will also need more cognitive scaffolding to help them learn new tools, languages and domains more rapidly. Pedagogical support, such as providing examples by analogy, will likely be an integral part of future software tools. The work discussed above on recommending code examples is also suggested at helping novices and software immigrants (i.e. programmers new to a project). Results from the empirical work also suggest that there is a need for tools to help programmers learn a new language. Technologies such as TXL [48] can play a role in helping a user see examples of how code constructs in one language would appear in a new language.

6.4 Discussion: *Back to the Future*

It is an interesting exercise to travel back and look at Tilley and Smith's paper from 1996 on "Coming Attractions in Program Understanding" [18]. Some of the predictions they had then for technologies which would be available within five years did mature as they predicted. They suggested that mature technologies would be leveraged, which is now the case with mature off the shelf technologies such as Windows' components and the Eclipse framework. They also predicted that web interfaces and hypertext would play a bigger role. Many modern tools now use web interfaces for navigating software resources. Tailorable user interfaces are now common place, as are more advanced pattern matching facilities.
Some of their predictions, however, are "still coming" and indeed overlap the predictions in this paper. These include: computer support collaborative understanding; access to alternative sources of data through natural language processing; data filters to allow the programmer to focus on relevant information; conceptual and domain modeling techniques; use of intelligent agents in a "maintainer's handbook" and more advanced visual interfaces. These suggestions are still under development but today seem within closer reach. Whether these themes will reappear in another 'prophecy' paper, written ten years from now, only time will tell.

7. Conclusions

As a community, we can be proud of our achievements over the past thirty years. As the landscape of comprehension research evolves, the future looks bright. We can anticipate that advances in program comprehension will increase, in part due to recent enabling technologies, such as sophisticated frameworks to support the more rapid construction and integration of tools, and advanced technologies such as context-aware and history-aware search and intelligent user interfaces. In parallel to these tool advances, there are now more researchers from both within computer science and from other disciplines that are interested in understanding more about the cognitive and social aspects of program comprehension. These researchers are now equipped with more appropriate research methods that can help to reveal more understanding about program comprehension needs and how these needs may be met through tool and process support. As the field of software engineering matures and the possibilities for more advanced and pervasive software increase, the field of program comprehension promises to be an exciting area for future research.

Acknowledgements

My thanks to Janice Singer, Hausi Muller, Robert Lintern, Adam Murray, and Jim Cordy for ideas, feedback and encouragement while writing this paper.

References

[1] NATO Software Engineering Conference, Garmisch, Germany, 7-11 Oct 1968.

[2] Brooks, Frederick P., "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, Vol. 20, No. 4, pp. 10-19, April 1987.

[3] Francoise Détienne, "Software Design -- Cognitive Aspects", Springer Practitioner Series, 2001.

[4] Luke Hohmann, "Journey of the Software Professional: The Sociology of Software Development", 1996.

[5] Andrew Walenstein, "Observing and Measuring Cognitive Support: Steps Toward Systematic Tool Evaluation and Engineering", 11th *Intl. Workshop on Program Comprehension* (IWPC'03), pp. 185-195, May 2003.

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

[7] Ruven Brooks, "Towards a theory of the comprehension of computer programs", *International Journal of Man-Machine Studies,* pp. 543-554, vol. 18, 1983.

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

[9] N. Pennington, "Stimulus structures and mental representations in expert comprehension of computer programs", *Cognitive Psychology,* pp. 295-341, vol 19, 1987.

[10] D.C. Littman, J. Pinto, S. Letovsky, and E. Soloway, "Mental models and software maintenance", In *Empirical Studies of Programmers*, pp. 80-98, 1986.

[11] S. Letovsky, "Cognitive processes in program comprehension", In *Empirical Studies of Programmer*s, pp. 58-79, 1986.

[12] A. von Mayrhauser and A.M. Vans, "From code understanding needs to reverse engineering tool capabilities", In *Proceedings of CASE'93*, pp. 230-239, 1993.

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

[14] Bill Curtis, "By the way, did anyone study any real programmers?", *Empirical studies of programmers*, pp. 256-262, 1986.

[15] T.J. Biggerstaff, B. W. Mitbander and D. Webster, "The concept assignment problem in program understanding", *Proceedings of the 15th international conference on Software Engineering*, pp. 482-498, 1993.

[16] J. Singer, T. Lethbridge, N. Vinson and N. Anquetil, "An Examination of Software Engineering Work Practices", *Proceedings of CASCON '97*, pp. 209-223, 1997.

[17] K. Erdös and H. M. Sneed, "Partial Comprehension of Complex Programs (enough to perform maintenance)", *Proceedings of the 6th International Workshop on Program Comprehension*, pp. 98-105, 1998.

[18] S. R. Tilley and D.B. Smith, "Coming Attractions in Program Understanding", Technical Report CMU/SEI-96-TR-019, 1996.

[19] H.A. Muller and K. Klashinsky, "Rigi: A system for programming-in-the-large", In *Proceedings of the 10th International Conference on Software Engineering (ICSE '10)*, pp. 80-86, April 1988.

[20] G.C. Murphy, D. Notkin and K. Sullivan, "Software Reflexion Models: Bridging the Gap Between Source and High-Level Models", In Proceedings of *Foundations of Software Engineering*, pp. 18-28, October 1995.

[21] T. Eisenbarth, R. Koschke and Daniel Simon, "Aiding Program Comprehension by Static and Dynamic Feature Analysis", In *Proceedings of the IEEE International Conference on Software Maintenance*, November 2001.

[22] M.-A. Storey, "Designing a Software Exploration Tool Using a Cognitive Framework of Design Elements", *Software Visualization*, Guest editor: Kang Zhang. Kluwer, March 2003.

[23] M. Lanza and S. Ducasse, "A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint", In *Proceedings of OOPSLA 2001*, pp. 300-311, ACM Press, 2001.

[24] A. Mockus, R. Fielding, and J.D. Herbsleb, "Two Case Studies of Open Source Software Development: Apache and Mozilla", *ACM Transactions on Software Engineering and Methodology*, 11, 3, pp. 309-346, 2002.

[25] C. Gutwin, R. Penner, and K. Schneider, "Group Awareness in Distributed Software Development", *ACM CSCW*, pp. 72 - 81, 2004.

[26] D. M. German, "Decentralized open source global software development, the GNOME experience", to appear in *Journal of Software Process: Improvement and Practice.*

[27] J. W. Creswell, "Research Design, Qualitative and Quantitative Approaches", SAGE Publications, 1994.

[28] M.-A. Storey, S.E. Sim and K. Wong, “A Collaborative Demonstration of Reverse Engineering Tools”, *ACM Applied Computing Review*, pp. 18-25, Spring 2003.

[29] C. Exton, “Constructivism and program comprehension strategies”, *10th International Workshop on Program Comprehensio*n, pp. 281–284, June 2002.

[30] T. R. G. Green and M. Petre, "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework", *Journal of Visual Languages and Computing,* 7(2), pp. 131-174, 1996.

[31] M. Kersten and G. Murphy, “Mylar: a degree-of-interest model for IDEs”, *International Conference on Aspect Oriented Software Development*, to appear, March 2005.

[32] J. Singer, R. Elves and M.-A. Storey, “NavTracks Demonstration: Supporting Navigation in Software Space”, submitted to *International Workshop on Program Comprehension*, 2005.

[33] R. DeLine, A. Khella, M. Czerwinski and G. Robertson, “Towards Understanding Programs through Wear-based Filtering”, *Softvis*, to appear May 2005.

[34] M. P. Robillard and G.Murphy, “FEAT: A tool for locating, describing, and analyzing concerns in source code”, In *Proceedings of the 25th International Conference on Software Engineering*, pp. 822-823, May 2003.

[35] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth, “Hipikat: A project memory for software development.” *IEEE Transactions on Software Engineering*, to appear in the special issue on mining software repositories.

[36] D. Mandelin, L. Xu, R. Bodik and D. Kimelman, “Mining Jungloids: Helping to Navigate the API Jungle”, PLDI, to appear, 2005.

[37] R. T. Holmes and G. C. Murphy, “Using structural context to recommend source code examples”, in *Proceedings of the International Conference on Software Engineering*, to appear, May 2005.

[38]] Storey, M.-A., J. Michaud, M. Mindel, M. Sanseverino, D. Damian, D. Myers, D. German and E. Hargreaves, “Improving the Usability of Eclipse for Novice Programmers”, *Eclipse Technology eXchange (eTX) Workshop* at OOPSLA 2003, October, 2003.

[39] T. Ball and S.G. Eick, “Software visualization in the large”, *IEEE Computer 29*, 4, pp.33-43, 1996.

[40] S. P. Reiss, “An overview of BLOOM”, *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pp.* 2-5, 2001.

[41] K. Wong, S.R. Tilley, H.A. Muller, and M.-A. Storey, “Structural redocumentation: A case study”, *IEEE Software*, 12(1), pp. 46-54, January 1995.

[42] D.A. Penny, “The Software Landscape: A Visual Formalism for Programming-in-the-Large”, *PhD thesis*, University of Toronto, 1992.

[43] A. Marcus, L. Feng, J.I. Maletic, "Comprehension of Software Analysis Data Using 3D Visualization", in *Proceedings of the IEEE International Workshop on Program Comprehension (IWPC 2003)*, pp. 105-114, May 2003.

[44] R. I. Bull and M.-A. Storey, “Towards Visualization Support for the Eclipse Modeling Framework”, *A Research-Industry Technology Exchange* at EclipseCon, March 2005.

[45] S. Hupfer, L.-T. Cheng, S. Ross and J. Patterson, “Introducing collaboration into an application development environment”, In *Proc. of the ACM Conference on Computer Supported Cooperative Wor*k, pp. 444-454, 2004.

[46] J. Froehlich and P. Dourish, “Unifying artifacts and activities in a visual tool for distributed software development teams”, In *Proc. of the 26th International Conference on Software Engineering*, pp. 387-396, 2004.

[47] C. O'Reilly, D. Bustard and P. Morrow, “The War Room Command Console [Shared Visualizations for Inclusive Team Coordination]”, Softvis, to appear 2005.

[48] J.R. Cordy, T.R. Dean, A.J. Malton and K.A. Schneider, "Source Transformation in Software Engineering using the TXL Transformation System", *Journal of Information and Software Technology,* vol(44)13, pp. 827-837, October 2002.

[49] K. Wong, “The Reverse Engineering Notebook”, *Ph.D. Thesis*, University of Victoria, 2000.