Technical Report
CMU/SEI-96-TR-019
ESC-TR-96-019

Coming Attractions in Program Understanding

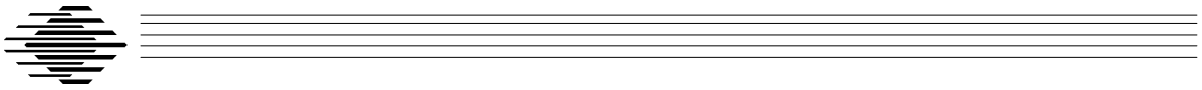
Scott R. Tilley

Dennis B. Smith

December 1996

Technical Report
CMU/SEI-96-TR-019
ESC-TR-96-019
December 1996

Coming Attractions in Program Understanding



Scott R. Tilley

Dennis B. Smith

Reengineering Center
Product Line Systems

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

This report was prepared for the

SEI Joint Program Office
HQ ESC/ENS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller,Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Research Access, Inc., 800 Vinial Street, Pittsburgh, PA 15212.
Phone: 1-800-685-6510. FAX: (412) 321-2994. RAI also maintains a World Wide Web home page. The URL is http://www.rai.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145. Phone: (703) 274-7633.

Table of Contents

List of Figures

Coming Attractions in Program Understanding

Abstract: *Program understanding* is the (ill-defined) deductive process of acquiring knowledge about a software artifact through analysis, abstraction, and generalization. This report identifies some of the emerging technologies in program understanding. We present technical capabilities currently under development that may be of significant benefit to practitioners within five years. Three areas of work are explored: investigating cognitive aspects, developing support mechanisms, and maturing the practice.

1. Introduction

This report identifies some of the emerging technologies in the area of program understanding that may to lead to advances that will be available to advanced practitioners of software engineering over the next five years. Most of these technologies are based on current research focus areas; a few are based on our judgment. Promising areas are those with potential to most positively affect software engineers in the next five years, either by providing revolutionary technologies that enable them to fully or partially automate previously manual tasks, or by enhancing their capabilities by improving the tools and techniques used to understanding programs.

Since this paper summarizes a changing field, it will be updated regularly. We invite your comments and suggestions for updated versions. Please contact us by sending e-mail to *stilley@sei.cmu.edu*

Increased knowledge of software aids in common activities such as performing corrective maintenance, reengineering, and keeping documentation up-to-date. Much current research focuses on ways to automate program understanding. However, significant amounts of domain knowledge, practical experience, and analytical power are required to achieve program understanding. These elements must usually be supplied by people.

Program understanding is critical in our ability to maintain and reengineer legacy systems. Many organizations are faced with maintaining aging software systems that are constructed to run on a variety of hardware types, are programmed in obsolete languages, and suffer from the disorganization that results from prolonged maintenance. As software ages, the task of maintaining it becomes more complex and more expensive. Software engineers must spend an inordinate amount of time creating representations of systems' high-level architecture from analyses of its low-level source code.

A legacy program may be inherently difficult to understand for several reasons. The complexity of the problem can make the solution (the program) complex. Poor design,

unstructured programming methods, and crisis-driven maintenance can contribute to poor code quality, which in turn affects understanding. Structuring mechanisms based on programming languages can create compatibility problems between the structure of the program and the structure of the users' mental model. In many cases, the only current, complete, and trustworthy information about a system is its source code; all other information must be derived from this.

Problem areas in program understanding can be grouped into the following three broad categories:

1. **Investigating cognitive aspects**. This category identifies how humans apply problem-solving techniques when attempting to understand a program. Topics include comprehension strategies, computer-supported cooperative understanding, and scenario-driven maintenance handbooks.
2. **Developing support mechanisms**. This category identifies how tools can be used to aid comprehension. Topics include alternative data-gathering techniques, advanced schemes for organizing knowledge, and hypertext-based information exploration.
3. **Maturing the practice**: This category identifies how emerging technology can mature so that it becomes part of the state-of-the-practice. Topics include making technology sufficiently robust that it is applicable to real-world problems, performing empirical studies to test the effectiveness of support mechanisms, and defining a universally accepted lexicon of terms.

We cover each of these categories in this report. Cross-cutting issues that affect these problem areas, such as scaleability, extensibility, and applicability, are discussed in Section 4, *Maturing the Practice*. Although these issues do not represent the three categories of program understanding work, they are extremely important.

2. Investigating Cognitive Aspects

The *cognitive aspects* of program understanding is the study of the problem-solving behavior of software engineers engaged in understanding tasks. Because the productivity of software engineers differs widely, analyzing the strategies used by those who are successful is most productive. This analysis leads to the development of tools and techniques that better support program understanding activities.

2.1 Comprehension Strategies

Software engineers employ *comprehension strategies* when they attempt to understand a program. To investigate comprehension strategies, you identify what information software engineers use to understand a software artifact and model how they use it. This is a fundamental area of research that crosses many disciplines, including software engineering, education, and cognitive science. A better understanding of comprehension strategies will enable the development of tools and techniques that provide greater support for program understanding than is currently possible.

Many theories have been formulated to explain the problem-solving behavior of maintainers and programmers engaged in program understanding. One survey of this area compared six cognitive models of program understanding [1]. Most cognitive models are usually variations on top-down understanding, bottom-up understanding, iterative hypotheses refinement, or some combination of the three. The bottom-up approach reconstructs the high-level design of a system, starting with source code, through a series of chunking[1] and concept-assignment steps. The top-down approach begins with a pre-existing notion of the functionality of the system and earmarks individual components of the system responsible for specific tasks. The iterative refinement approach creates, verifies, and modifies hypotheses until the entire system is explained by a consistent set of hypotheses. The combination approach opportunistically exploits top-down and bottom-up cues as they become available.

Studies show that maintainers regularly switch between these different appoachs depending on the problem-solving task at hand. Consequently, no single model explains all program understanding behavior, even though some models encompass other models. By investigating comprehension strategies that better reflect the actual understanding approaches used by expert software engineers, identifying when specific comprehension approaches are best used will become clearer.

[1] Chunking refers to mentally clustering logically related source code fragments together.

2.2 Computer-Supported Cooperative Understanding

As software systems grow in size and in complexity, teams of software engineers performing maintenance tasks together will become increasingly common. This trend means that the traditional approach to program understanding, which has focused on a single person working in isolation, must evolve to support cooperative understanding.

A software system that provides a shared environment to support groups of people engaged in a common task is sometimes called “groupware”. A related term is computer-supported cooperative work (CSCW), which combines a study of the organizational, psychological, and social aspects of people working together with the enabling technologies of groupware [2]. Much work in CSCW is currently focused on using the web as an infrastructure to support geographically distributed collaborative efforts.

Applying CSCW techniques to program understanding can be called *computer-supported cooperative understanding* (CSCU). Although still in its infancy, CSCU shows promise in program-understanding tasks that require the effort of more than one software engineer. CSCU enables software engineers with diverse areas of expertise to combine their knowledge so they can better analyze and modify legacy systems. CSCU also enables software engineers to work in an asynchronous and location-independent manner. Small teams of perhaps three to five people can work together to solve a program understanding problem that may be too complex for a single person to solve on their own.

2.3 Maintenance Handbooks

Many engineering disciplines make extensive use of handbooks to guide practitioners in solving well-understood problems. For example, electronics technicians have reference handbooks to help them while repairing complex circuitry. Problem symptoms are often used as an index into the handbook, where solutions that are known to work for the problem are explained. The explanation usually includes the tools required to implement the solution. Such handbooks capture expert strategies that help others who lack the same experience.

Experience seems to be a significant factor in successfully understanding a program. Experienced programmers tend to have superior programming knowledge, make better use of tools, and employ systematic comprehension strategies to make the task easier. They also tend to have valuable domain expertise that can be used to confront new tasks that resemble previous ones. Instead of concentrating on how the program works (as a less experienced person might), a more experienced software engineer tends to form representations of “what the program does.” They attempt to reuse knowledge and expertise from previous tasks by first looking for a link between what they perceive and an existing model structure before conceiving a new model.

As we gain more knowledge about how people understand programs by developing cognitive models and validating them in comprehension experiments, this knowledge should be codified. This codified knowledge could take the form of *maintenance handbooks* that capture the expertise and strategies proven effective for general and specific maintenance scenarios. Such handbooks would provide practitioners with prescriptive solutions to common problems.

Maintenance handbooks could evolve to take the form of intelligent agents that aid software engineers in program understanding tasks. Similar technology is already used in commercial software to aid users in performing common business tasks, such as writing a letter. While program understanding tasks are generally more complex and less well-defined that writing a letter, the use of agents for specific, high-level tasks would be beneficial because they could automate some parts of the problem-solving excercise.

3. Developing Support Mechanisms

One way of helping software engineers with program-understanding tasks is through computer-aided *support mechanisms*. Such tools and techniques can manage the complexities of program understanding by helping the software engineer extract high-level information from low-level code. These support mechanisms free software engineers from tedious, manual, and error-prone tasks such as code reading, searching, and pattern-matching by inspection.

Reverse engineering is a particularly important type of program understanding support mechanism. Reverse engineering is seen as an activity that does not change the subject system; it is a process of examination, not a process of alteration. It aids program understanding by helping identify artifacts, discover their relationships, and generate abstractions. The following discussion categorizes the support mechanisms into three canonical reverse engineering activities [3]: (1) data gathering, (2) knowledge organization, and (3) information exploration (includes navigation, analysis, and presentation).

So far, reverse engineering has been relatively successful in aiding program understanding. It is more robust and scaleable than pure, artificial intelligence-based, automated program understanding tools, easier to use than formal methods based on theorem-proving, and more attractive than non-computer-aided techniques such as code reading for very large source codes. Support mechanisms can fit into one (or more) of the three reverse engineering activity areas.

3.1 Data Gathering

To identify the artifacts and relationships of a system and use them to later construct and explore higher-level abstractions, you must gather raw data. Hence, *data gathering* is an essential reverse engineering activity. New developments in data gathering techniques benefit practitioners by providing them with more accurate and extensive capabilities they can use to extract artifacts of interest from their programs. Because data represents the building blocks upon which more abstract representations of the legacy system are built, it is critically important that the data gathered not be misleading or subject to misinterpretation; it must be factual and objective.

3.1.1 Using the Leverage of Mature Technology

The predominant technique used for gather data is parsing a system's source code to construct abstract syntax trees with the large number of fine-grained syntactic artifacts and dependencies. To accomplish this, many researchers have spent an inordinate amount of time building parsers for various programming languages and dialects. However, mature technology already exists in the compiler arena that will parse source code, perform

syntactical analysis, and produce cross-reference and other information that can be used by other tools, such as debuggers.

By using the leverage of proven compiler-based technology for data gathering, users of reverse engineering tools will be assured of predictable results. This is not currently the case: there are several extraction tools that, when applied to the same source code, produce somewhat different results [4]. Practitioners and researchers alike will benefit greatly once traditional tools, such as compilers, are integrated in newer program-understanding toolsets. This will produce data that is more trusted and accurate.

3.1.2 Alternative Sources of Data

In addition to using data gathered from traditional sources, such as compiler-based static analysis, work is currently underway to integrate *alternative sources of data* into reverse engineering toolsets. Examples include dynamic analysis (for example, profiling), natural-language content analysis (for example, from comments and/or other documentation, and source code naming conventions), and informal data extraction (for example, interviewing). These non-traditional techniques can provide a basis for a more balanced and complete understanding of programs by emphasizing different attributes of program artifacts and relationships. This especially can benefit software engineers who work with programs that are difficult to understand when using only data gathered through static source code analysis. For example, dynamic analysis provides data that can aid the understanding of distributed, real-time, client-server programs (applications that are becoming increasingly predominant, and hence will shortly become legacy systems themselves).

In-line comments[2] are a potentially rich source of data about the program, and are often used by experts when attempting to understand a software artifact. However, automatic analysis of in-line comments and other written commentary, such as program logic manuals, is more difficult. Techniques such as natural language analysis are needed to parse these comments. In addition, judgment must be used to link comments to the code it purports to describe. Comments may be isolated in the code, or (even worse) they may no longer reflect reality and may provide conflicting information if the comments were not updated with the code. Nevertheless, comments represent such a potentially rich data source that work continues to focus on their analysis.

Another source of data about software programs its human maintainers. Interviewing techniques can be used to capture the expertise of such people. This “corporate knowledge” is a potentially valuable asset if it can be applied to program understanding.

[2] Comments written in the same file as the source code.

3.1.3 Data Filtering

No matter what the source, the amount of data gathered for understanding large systems can be enormous. Large quantities of data can easily overwhelm our ability to assimilate it. Therefore, the use of intelligent *data filtering* techniques play an important role in aiding program understanding. Presenting the user with reams of data is insufficient. To understand the data, the user must also assimilate the data. In a sense, a key to program understanding is deciding what is material and what is immaterial. In other words, knowing what to look for and what to ignore [5].

Data filters can be used to extract selected artifacts and relationships from a rich data source. For example, a profiling tool may be used to gather complete run-time call information from a program, but the software engineer may be interested in only a subset of these calls. Such filters can also be used as an interface between tools that do not share a common data representation.

3.2 Knowledge Organization

For successful program understanding, data must be in a form that facilitates efficient storage and retrieval, permits analysis of artifacts and relationships, and reflects the users' perception of the system's characteristics. This form is usually based on a data model. A data model enables us to understand the essential properties and relationships between artifacts in a system. Without a model, raw data is almost impossible to understand. We rely on *knowledge-organization* techniques to create, represent, and reason about data models.

3.2.1 Advanced Modeling Techniques

Classical physical data models, such as the hierarchical, network, and relational models, capture data and their relationships in a form best suited to computer manipulation. In contrast, *advanced modeling techniques* capture data and their relationships in a form best suited to human understanding. These newer modeling techniques provide abstraction mechanisms that help the software engineer organize knowledge about the subject system.

One advanced modeling technique is *conceptual modeling*, which is closer to a human understanding of a problem domain than to a computer representation of the problem domain [6]. This technique emphasizes knowledge organization (modeling entities and their semantic relationships) rather than data organization. The descriptions resulting from conceptual modeling are intended for use by people—not machines. Because it is designed for people, conceptual modeling is eminently suited to aiding program understanding.

3.2.2 Iterative Domain Modeling

Domain modeling is the process of identifying, organizing, and representing the structure and composition of elements in a problem area. This type of modeling can be to help organize knowledge about a subject system. The construction of the domain model can precede reverse engineering (so it can be used to guide the understanding process by supplying expected constructs), or it can be constructed during reverse engineering (if no previous knowledge about the domain was available). Hence, a domain model can be *a guide to* and *a product of* reverse engineering, or it can be combined into *iterative domain modeling* to support exploratory understanding [7]. The software engineer benefits by gaining a new method of tool-assisted program understanding. This new method also enables the software engineer to use tools that automatically recognize standard components of a system and use these components to populate the domain model. The software engineer can also use semi-automatic or manual techniques, as part of this new method, to classify non-standard components and use this information to extend the domain model.

3.2.3 Scaleable Knowledge Bases

As mentioned earlier, the volume of data produced during the reverse engineering of a large-scale software system is considerable. Such size and complexity require *scaleable knowledge bases* that use fundamentally different approaches to repository technology than is used in other application domains. For example, not all software artifacts need to be stored in the repository; some artifacts may be ignored. Coarse-grained artifacts can be extracted, partial systems can be incrementally investigated, and irrelevant parts can be ignored to obtain manageable repositories. Once available, scaleable knowledge bases will enable improved understanding of large software systems.

Scaleable knowledge bases will also need to support CSCU (as discussed in Section 2.2). This includes access to a central repository in a distributed environment, and synchronization with workspaces holding the results of local analyses. Dynamic schema evolution would facilitate this.

3.3 Information Exploration

Because the majority of program understanding takes place during *information exploration*, it is perhaps the most important of the three canonical reverse engineering activities. Data gathering is required to begin the reverse engineering process. Knowledge organization is needed to structure the data into a conceptual model of the application domain. The key to increased comprehension is exploration because it facilitates the iterative refinement of

hypotheses. Exploration includes navigating through the hyperspace[3] that represents the information related to the subject system, analyzing and filtering this information with respect to domain-specific criteria, and using presentation mechanisms to clarify the resultant information.

3.3.1 Navigation

Large software systems, like other complex systems, are non-linear and may be viewed as consisting of an interwoven and multidimensional web of information artifacts. The web's links establish relationships between the artifacts. These relationships can be component hierarchies, inheritances, data and control flow, and other relationships generated as part of the reverse engineering process. *Navigation* allows software engineers to traverse this "information web" as part of their exploratory understanding activities.

Reducing disorientation. As the size of this web grows, the well-known "lost in hyperspace" syndrome limits navigational efficiency. Several strategies are being investigated to meet the classic challenge of *reducing disorientation* within a large information space. These strategies include maps, multiple windows, history lists, and tour/path mechanisms [8]. Unfortunately, many of these methods are not sufficiently scaleable.

A more promising strategy is the use of composite nodes. These nodes reduce web complexity and simplify its structure by clustering nodes together to form more abstract, aggregate objects [9]. Composite nodes interact with sets of nodes as unique entities, separate from their components. Navigation tools that support such clustering will be very useful in program understanding.

WWW-based interfaces. Notwithstanding the disorientation challenge discussed above, hypertext-based navigation can enable the software engineer to choose and deploy navigation strategies that are most suitable to the task at hand. There are several current reverse-engineering systems that employ a hypertext user interface. However, such interfaces generally remain proprietary and require users to learn new interaction methods and tools to use them effectively. Work is underway to develop these hypertext-based systems to use more generally-available graphical user interfaces, most notably browsers for the World Wide Web (WWW). The explosive growth of both the Internet and the WWW will make it possible for program understanding technology to be delivered to practitioners in a familiar form (*WWW-based interfaces).* This delivery mechanism will also enable practitioners to integrate reverse-engineering support mechanisms no matter where their actual installation location is.

[3] The structured information space composed of "nodes" representing artifacts from the application domain and "links" representing relationships between the nodes. Sometimes referred to as a "web".

Advanced pattern matching. Pattern matching is an essential part of program understanding. Locating relevant code fragments that implement the concepts in the application domain requires much effort. Reverse engineering involves the identification, manipulation, and exploration of artifacts in a particular representation of the subject system using pattern recognition. This pattern recognition is accomplished either mentally by the software engineer, or mechanically by the support mechanism. Artifacts are segmented into features. The patterns of these features are then matched against stored collections of expected structural motifs. The success of this process depends on the recollection of existing structural knowledge and on the ability of the person (or tool) to recognize its presence in a noisy environment.

An emerging trend is the development of *advanced pattern-matching* techniques that concentrate more on the *meaning* of the code rather than on its *form*. These techniques will enable the software engineer to reduce the amount of time and effort spent switching between domains (for example, from the application domain to the implementation domain) during program understanding. If the patterns can be represented in terms related to the application domain (where most change requests are couched), then the software engineer can more easily change the source code with fewer surprises.

3.3.2 Analysis

Analysis is the critical step that derives abstractions from the raw data. Software engineers use the resultant information to further understand the system. There are many new analysis techniques currently under investigation that may have significant impact on practitioners in a few years. One example is *slicing* [10]. This analysis technique identifies program code fragments that may affect the value of selected variables. By isolating the statements that can change the value of a variable (or variables), the cognitive overhead of understanding a large piece of code is reduced significantly.

End-user programmability. Rather than limiting software engineers to designer-defined analyses that are invoked using canned methods, it is better to provide mechanisms that programmers can use to define their own analyses. Some leading-edge reverse-engineering systems aid program understanding by providing full-fledged programming languages that can be used to encode analysis methods. Software engineers will be able to use *end-user programmability* in reverse engineering support mechanisms to develop analysis techniques for specific tasks. This flexibility will increase the likelihood that analyses will better apply to unique software systems.

Automation level. It is important to manage the tradeoff between the functions that are handled automatically by a reverse engineering tool and the functions that enable the tools to accept human input and guidance. Current work is focused on how to best balance between automatic, semi-automatic, and manual approaches, where each is more applicable, and how the support mechanism can “know” when to ask for expert guidance. Using the correct *automation level* can affect both the time taken to complete a program-

understanding task, and the level of comprehension achieved. Within the next five years we will likely see more automation of program understanding tasks as they become better understood (see Section 2.3 on maintenance handbooks).

Higher-order impact analysis. Estimating the effect of changes before they are irrevocable has always been an important part of program understanding. Engineers try to avoid causing massive changes to a system during maintenance. Their avoidance is due, in part, to practical issues such as recompilation delays, but more importantly because they are unwilling to create “change waves” that ripple throughout large parts of the system. The potential for errors caused by these waves is too great. Current tools perform impact analysis primarily at the syntactic level. Newer research, however, focuses on *higher-order impact analysis* tools that allow users to perform “what if” scenarios and analyze the result of proposed changes. These higher-order impact analysis tools will enable the software engineer to function at the application-domain level than the implementation-domain level.

3.3.3 Presentation

Because people often use visual metaphors for communicating and understanding information, it is important to use flexible *presentation* mechanisms. Currently, most reverse-engineering systems provide the user with fixed presentation options, such as cross-reference graphs or module-structure charts. Even though the producers of the system might consider fixed options adequate, there are always users who want something else. Ideally, it should be possible to create multiple, perhaps orthogonal, structures and to view them using a variety of mechanisms (for example, different graph layouts provided by external toolkits).

Advanced visualization techniques. Graph layout theory has already proven effective in aiding program understanding. For example, graphical representations of source code proliferate in current reverse-engineering systems. Refinements to this traditional area also show promise. An example is so-called “fish-eye” views that emphasize on selected focal points while retaining relative location information [11]. Exploratory work is underway on more *advanced visualization techniques* using three-dimensional data imaging, virtual reality “code walk-through," and user-defined views. One or more of these techniques may provide new insights into program understanding.

Tailorable user interfaces. Presentation integration can occur at different levels, including the window manager, the toolkit used to build applications, and the toolkit’s “*look and feel*” [12]. The standardization provided by presentation integration lessens the cognitive surprise experienced by users when switching between tools. However, what is really needed is a *tailorable user interface* that permits users to impose their own personal *taste* on the common look and feel. This refinement of presentation integration moves the onus, and the opportunity, for reducing complexity of the user interface from the tool builder to the tool user.

Multimedia. Presentation of analysis results has traditionally taken the form of charts, tables, or graphs. The recent proliferation of *multimedia*-enhanced computers introduces new ways of presenting this information. An area that shows promise is the use of audio and video annotations as a way of commenting source code, capturing programmer rationale, and presenting information to the user in more familiar and readily accessible ways.

4. Maturing the Practice

Until recently, most work in program understanding focused on “toy” programs that barely resembled real-world legacy systems. Fortunately, this has begun to change. As the state-of-the-art *matures* and becomes the state-of-the-*practice*, promising work is focusing on easing the insertion of program understanding technology, on performing empirical studies, and on creating a common lexicon of related terms. Practitioners will benefit because program understanding tools and techniques will be easier to use, more applicable to everyday tasks, and more widely perceived as a legitimate technology capable of solving real-world maintenance problems.

4.1 Technology Insertion

For program-understanding technology to have more impact and gain widespread use, it must address several transition issues, including scaleability, extensibility, and applicability. Only after addressing these issues can *technology insertion* take place on a larger scale. Inserting program understanding technology into current development and maintenance processes will benefit users by making new capabilities available in a more uniform way. As a result, using new technologies (such as slicing) will become as commonplace as established techniques (such as browsing cross-reference charts).

Legacy software systems can contain millions of lines of source code. Because of this, support mechanisms need to be sufficiently robust to function effectively at this scale. Increasing the extensibility of support mechanisms through end-user programmability, by using techniques such as scripting or macro languages, make the integration of program-understanding tools with the practitioner’s existing tools easier. Such extensibility also enables the use of current reverse-engineering techniques (which are sometimes narrowly focused) for more general applications.

As repeatedly shown in other fields (for example, CASE), a new technology is only successful if it integrates with existing tools and processes. Forcing users to adopt radically new ways of working rarely succeeds. Only through pragmatic technology insertion can program understanding become less of a novelty and more of an accepted practice.

4.2 Empirical Studies

Experimentation plays an important role in both the theory (investigating cognitive aspects) and practice (developing support mechanisms) of program understanding. Without *empirical studies*, there is no systematic way to validate or refute hypotheses or claims of functionality. With empirical studies, practitioners can make better and more informed decisions about how well certain program-understanding technologies apply to their specific problems.

Most engineering disciplines conduct empirical studies as initial investigation or as theory validation. The sophistication of the empirical studies reflects the maturity of an engineering field. While empirical studies have been done for performing program-understanding experiments, much more work is needed. There is a clear need for a common test suite: a subject system (or systems) that can be used to perform metrics. The continued maturation of the field depends on the performance of scientific, verifiable, and repeatable empirical studies. Such studies provide software engineers with independent and indisputable proof of the gains possible using program-understanding technology and reduce their reliance on unsubstantiated claims.

Practitioners also stand to benefit from participating in early-adoption trials and case studies. Such experiments can be win-win situations for researchers and tool developers as well as tool users. Researchers and tool developers validate, strengthen, or refocus their work on practical problems. Users gain hands-on experience with the technology before others and can guide development of the technology to be more beneficial to their daily work.

4.3 Common Terminology

Established fields of study usually have a common vocabulary with agreed-upon meanings for key phrases. As the program-understanding community matures, it too will develop *common terminology* that will aid researchers and practitioners alike. Clear exchanges of information will ensure that one of the hallmarks of a developing field will be addressed: confusion over the meaning of new terminology.

A seminal paper in reverse engineering by Chikofsky and Cross appeared in the January 1990 issue of IEEE Software [13]. It is the first attempt to create a taxonomy of concepts related to program understanding. The paper identifies often-cited definitions of “reverse engineering” and many related terms. Unfortunately, different interpretations of “reverse engineering” and other related phrases remain. For example, there are several “near-synonyms” for “reengineering.” This fact illustrates the relative immaturity of the field’s nomenclature. Work is continuing on refining and expanding this early lexicon.

Practitioners will benefit from common terminology by forcing vendors to use the same words to represent the same functionality. For example, if a tool claims to extract “business rules” from source code, there must be a clear definition of what a business rule is in this context and how it is represented. The move toward common terminology represents a development in program understanding that is long overdue.

5. Summary

This paper presented “coming attractions” in program understanding. These coming attractions are improved technical capabilities that are under development and could be ready for evaluation and demonstration in the next five years. Three promising lines of research were discussed: investigating cognitive aspects, developing support mechanisms, and maturing the practice. These new capabilities have the potential to make a significant positive impact on practicing software engineers who are looking ahead.

Figure 1 summarizes improved capabilities under the three promising lines of emerging technologies. An estimate of when these three developments can be expected, given the current trends, is included. These time estimates are educated guesses and may be strongly affected by factors such as funding cuts, market shifts, and technological breakthroughs. The purpose of the five-year time frame is to stimulate discussion about likely paths to achieve the goals. These predictions will be updated periodically to reflect new trends and developments.

Acknowledgments

Thanks to John Salasin of DARPA, Paul Clements of SEI, Alan Brown of Texas Instruments, Spencer Rugaber of Georgia Institute of Technology, and Bob Moore of Blair & Associates for providing comments on the early drafts of this paper. This work was supported in part by the U.S. Department of Defense.

Emerging Technology	Early Availability				
	1996	1997	1998	1999	2000
1 **Investigating cognitive aspects**					
Comprehension strategies	x				
CSCU			x		
Maintenance handbooks			x		
2 **Developing support mechanisms**					
Data gathering					
Leveraging mature technology		x			
Alternative sources of data		x			
Data filtering	x				
Knowledge organization					
Advanced modeling techniques				x	
Iterative domain modeling					x
Scaleable knowledge bases			x		
Information exploration					
Navigation					
Reduced disorientation		x			
WWW-based interfaces	x				
Advanced pattern matching				x	
Analysis					
End user programmability	x				
Automation levels		x			
Higher-order impact analysys				x	
Presentation					
Advanced visualization techniques			x		
Tailorable user interfaces		x			
Multimedia			x		
3 **Maturing the practice**					
Technology insertion			x		
Empirical studies		x			
Common terminology			x		

Figure 1: A Possible Timeline for Coming Attractions in Program Understanding

References

[1] A. v. Mayrhauser and M. Vans, “Program Comprehension During Software Maintenance and Evolution,” *IEEE Computer*, vol. 12, pp. 44-55, 1995.

[2] M. Koch, “Introduction to CSCW,” Technical University of Munich, Germany http://www11.informatik.tu-muenchen.de/cscw/, 1996.

[3] S. R. Tilley, S. Paul, and D. B. Smith, “Towards a Framework for Program Understanding,” presented at 4th Workshop on Program Comprehension, Berlin, Germany, 1996.

[4] G. C. Murphy, D. Notkin, and E. S.-C. Lan, “An Empirical Study of Static Call Graph Extractors,” presented at 18th International Conference on Software Engineering, Berlin, Germany, 1996.

[5] M. Shaw, “Larger Scale Systems Require Higher-Level Abstractions,” *ACM SIGSOFT Software Engineering Notes*, vol. 14, pp. 143-146, 1989.

[6] B. B. Kristensen and K. Osterbye, “Conceptual Modeling and Programming Languages,” *ACM SIGPLAN Notices*, vol. 29, 1994.

[7] J.-M. DeBaud, B. M. Moopen, and S. Rugaber, “Domain Analysis and Reverse Engineering,” presented at International Conference on Software Maintenance, Victoria, British Columbia, Canada, 1994.

[8] J. Nielsen, *Hypertext and Hypermedia* Academic Press, 1990.

[9] M. A. Casanova, L. Tucherman, M. J. D. Lima, J. L. R. Netto, N. Rodriguez, and L. F. G. Soares, “The Nested Context Model for Hyperdocuments,” presented at Hypertext '91, San Antonio, TX, 1991.

[10] M. Weiser, “Program Slicing,” *IEEE Transactions on Software Engineering*, vol. SE-10, pp. 352-357, 1984.

[11] M. A. D. Storey and H. A. Muller, “Graph Layout Adjustment Strategies,” presented at Graph Drawing '95, Passau, Germany, 1995.

[12] A. I. Wasserman, “Tool Integration in Software Engineering Environments,” presented at International Workshop on Environments, Chinon, France, 1989.

[13] E. J. Chikofsky and J. H. C. II, “Reverse Engineering and Design Recovery: A Taxonomy,” *IEEE Software*, vol. 7, pp. 13-17, 1990.