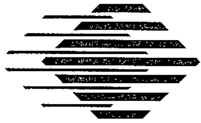


Carnegie Mellon University
Software Engineering Institute

A Reverse-Engineering Environment Framework

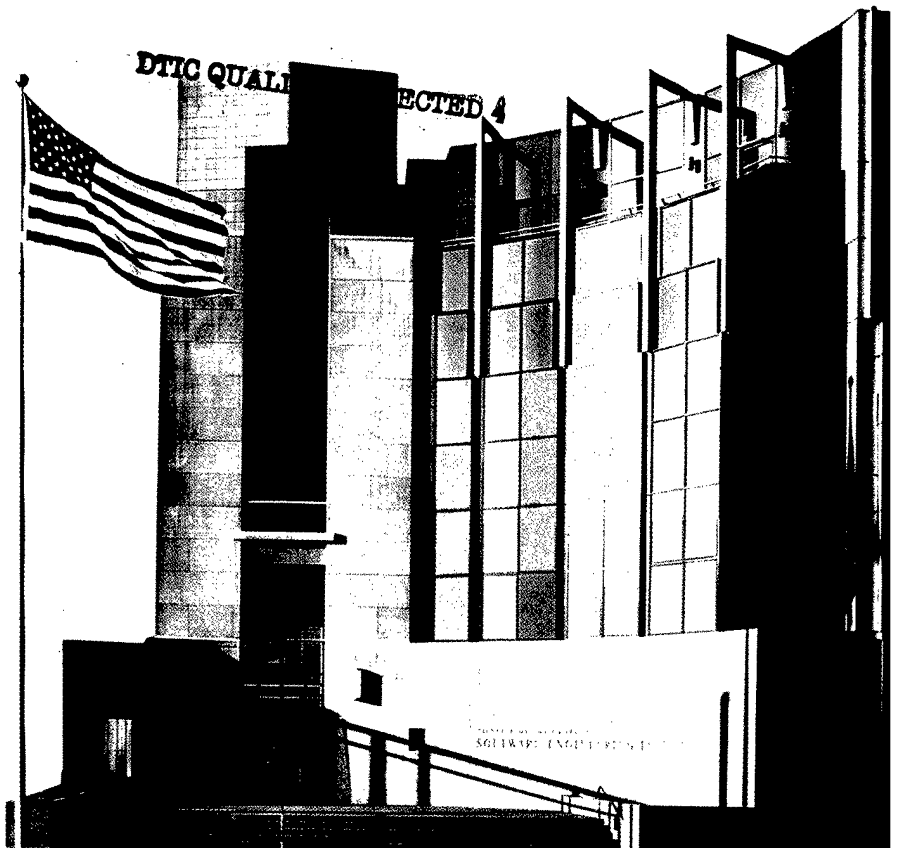
Scott Tilley

April 1998

19980514 115

TR

TECHNICAL REPORT
CMU/SEI-98-TR-005
ESC-TR-98-005



DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

Carnegie Mellon University does not discriminate and Carnegie Mellon University is required not to discriminate in admission, employment, or administration of its programs or activities on the basis of race, color, national origin, sex or handicap in violation of Title VI of the Civil Rights Act of 1964, Title IX of the Educational Amendments of 1972 and Section 504 of the Rehabilitation Act of 1973 or other federal, state, or local laws or executive orders.

In addition, Carnegie Mellon University does not discriminate in admission, employment or administration of its programs on the basis of religion, creed, ancestry, belief, age, veteran status, sexual orientation or in violation of federal, state, or local laws or executive orders. However, in the judgment of the Carnegie Mellon Human Relations Commission, the Department of Defense policy of, "Don't ask, don't tell, don't pursue," excludes openly gay, lesbian and bisexual students from receiving ROTC scholarships or serving in the military. Nevertheless, all ROTC classes at Carnegie Mellon University are available to all students.

Inquiries concerning application of these statements should be directed to the Provost, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-6684 or the Vice President for Enrollment, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-2056.

Obtain general information about Carnegie Mellon University by calling (412) 268-2000.

CMU/SEI-98-TR-005
ESC-TR-98-005

A Reverse-Engineering Environment Framework

Scott Tilley

April 1998

Reengineering Center
Product Line Systems

Carnegie Mellon
Software Engineering Institute
Pittsburgh, PA
15213-3890

Unlimited distribution subject to the copyright.

This report was prepared for the

SEI Joint Program Office
HQ ESC/DIB
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

Jay Alonis, Lt Col, USAF
SEI Joint Program Office

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

Copyright 1998 by Carnegie Mellon University.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Internal use. Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and "No Warranty" statements are included with all reproductions and derivative works.

External use. Requests for permission to reproduce this document or prepare derivative works of this document for external and commercial use should be addressed to the SEI Licensing Agent.

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 Asset Source for Software Engineering Technology (ASSET): 1350 Earl L. Core Road; PO Box 3305; Morgantown, West Virginia 26505 / Phone: (304) 284-9000 or toll-free in the U.S. 1-800-547-8306 / FAX: (304) 284-9001 World Wide Web: http://www.asset.com / e-mail: sei@asset.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: BRR / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218 / Phone: (703) 767-8274 or toll-free in the U.S.: 1-800 225-3842.

Table of Contents

Abstract v

1. Introduction 1

1.1 Program Understanding Support Mechanisms 1

1.2 About the Reverse-Engineering Environment Framework 2

1.3 Organization of This Report 3

2. Cognitive Model Support 5

2.1 Bottom Up 5

2.2 Top Down 5

2.3 Opportunistic 6

3. Reverse-Engineering Tasks 7

3.1 Program Analysis 7

3.2 Plan Recognition 8

3.3 Concept Assignment 9

3.4 Redocumentation 9

3.5 Architecture Recovery 10

4. Canonical Activities 11

4.1 Data Gathering 11

4.1.1 System Examination 12

4.1.2 Document Scanning 13

4.1.3 Experience Capture 14

4.2 Knowledge Management 14

4.2.1 Organization 14

4.2.2 Discovery 17

4.2.3 Evolution 18

4.3 Information Exploration 19

4.3.1 Navigation 19

4.3.2 Analysis 21

4.3.3 Presentation 23

5. Quality Attributes 25
5.1 Applicability 25
5.1.1 Application Domain 26
5.1.2 Implementation Domain 26
5.2 Extensibility 27
5.2.1 Integration Mechanisms 27
5.2.2 End-User Programmability 28
5.2.3 Automatability 28
5.3 Scalability 29

6. Miscellaneous Characteristics 31
6.1 Computing Platform 31
6.2 Ancillary Requirements 31
6.3 Cost 31

7. Summary 33

Acknowledgements 35

References 37

List of Figures

Figure 1: Pattern Abstraction for Levels of Program Analysis 7

Figure 2: Application Domains and Support Mechanism Requirements 26

Figure 3: Implementation Domains and Support Mechanism Requirements 27

Figure 4: The Descriptive Model 34

Abstract

This report describes a framework for reverse-engineering environments used to aid program understanding. The framework is based on a descriptive model that categorizes important support mechanism features based on a hierarchy of attributes. The attributes include cognitive model support, reverse-engineering tasks, canonical activities that are characteristic of the reverse-engineering process, quality attributes supported by the reverse-engineering environment, and miscellaneous characteristics.

1. Introduction

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. Poor design, unstructured programming methods, and crisis-driven maintenance can contribute to poor code quality, which in turn affects understanding.

Better understanding of a program aids in common activities such as performing corrective maintenance, reengineering, and keeping documentation up to date. To minimize the likelihood of errors introduced during the change process, the software engineer must understand the system sufficiently well so that changes made to the source code have predictable consequences. But such understanding is difficult to recover from a legacy system after many years of operation.

Program understanding is a relatively immature field of study in which the terminology and focus are still evolving. The goal of program understanding is to acquire sufficient knowledge about a software system so that it can evolve in a disciplined manner. The essence of program understanding is identifying artifacts and understanding their relationships; this process is essentially pattern matching at various abstraction levels. It involves the identification, manipulation, and exploration of artifacts in a particular representation of a subject system via mental pattern recognition by the software engineer and the aggregation of these artifacts to form more abstract system representations.

1.1 Program Understanding Support Mechanisms

There are a variety of support mechanisms for aiding program understanding. They can be grouped into three categories: unaided browsing, leveraging corporate knowledge and experience, and computer-aided techniques like reverse engineering. Unaided browsing is essentially "humanware": the software engineer manually flips through source code in printed form or browses it online, perhaps using the file system as a navigation aid. This approach is almost always used in some form, but it is not really a viable approach for very large systems. A good software engineer may be able to keep track of approximately 50,000 lines of code in his or her head. If there is much more than that, then the amount of information to keep track of becomes unwieldy.

The second category of support mechanism is leveraging corporate knowledge and experience. This can be done through mentoring or by conducting informal interviews with person-

nel knowledgeable about the subject system. This approach can be very valuable if there are people available who have been associated with the system as it has evolved over time. They carry important information in their heads about why the system was designed the way it was, the major changes that have occurred over its life cycle, and where subsystems have proven particularly troublesome. For example, they may be able to provide guidance on where to look when carrying out a new maintenance activity if it is similar to another change that took place in the past. This approach is useful both for gaining a big-picture understanding of the system and for learning about selected subsystems in detail. Unfortunately, this type of corporate knowledge and experience is not always available. The original designers may have left the company. The software system may have been acquired from another company. Or the system may have had its maintenance out-sourced.

In this situation, the only recourse is the third category of support mechanisms: computer-aided *reverse engineering*. A reverse-engineering environment can manage the complexities of program understanding by helping the software engineer extract high-level information from low-level artifacts, such as source code. This frees software engineers from tedious, manual, and error-prone tasks such as code reading, searching, and pattern matching by inspection.

1.2 About the Reverse-Engineering Environment Framework

Although substantial process has been made in tool-based environmental support for aiding program understanding, there is not a satisfactory mechanism to classify the technology that is currently available. As a result, it is difficult to compare the purposes, functionality, and characteristics of different program-understanding tools and techniques. To address this need, this report provides a descriptive model that categorizes important support mechanism features based on a hierarchy of attributes.

The model can be used for characterizing an individual support mechanism, a set of which can then be compared using a common vocabulary. At present, the model organizes attributes according to the broad categories of cognitive-model support, reverse-engineering tasks, canonical activities, quality attributes, and miscellaneous characteristics.

The reverse-engineering environment framework described in this report is based on an earlier effort that was called "Towards a Framework for Program Understanding" [Tilley 96a]. This earlier work was an initial attempt to solicit feedback from the community on the structure and contents of the framework. Two special meetings were held to discuss the framework. The first was during the 1996 Workshop on Program Comprehension and the second was during the 1996 Software Technology Conference. One of the most important comments received was that the framework was too top-down. It was unclear whether the framework was meant to characterize research efforts or to characterize reverse-engineering environments. Since the primary use of the framework is to guide advanced practitioners on reverse

engineering options, the framework was reorganized to reflect this goal. A new effort, the Program Understanding Framework, was started to characterize current activity areas in program understanding. This latter framework is also under revision and has been used to describe the current state-of-the-practice in program understanding [Tilley 96b, Tilley 98].

Since 1996 the reverse-engineering environment framework has received input via email based on the material on the Reengineering Center's Web site (http://www.sei.cmu.edu/reengineering). In addition, the framework has benefited enormously from discussions with selected representative of academia and industry. It is expected that further meetings with various members of the program understanding, reverse engineering, and reengineering communities will continue to contribute to the framework as it evolves.

1.3 Organization of This Report

The next section discusses support for different cognitive models that can greatly affect the usefulness of a reverse-engineering environment. Section 3 describes the typical reverse-engineering tasks; whether or not an environment supports these tasks can be a motivating factor for selecting one environment over another. Section 4 describes the canonical activities that are characteristic of any reverse engineering task, no matter what environment is used. A reverse-engineering environment also exhibits certain quality attributes (the "ilities") that affect its usefulness. For example, the degree of extensibility of the system can affect how well the tool can be tailored to specific reverse-engineering tasks. Section 5 explores some of these quality attributes in more detail. Section 6 discusses the miscellaneous characteristics that can be important factors in the selection of a reverse-engineering environment, such as cost. Section 7 summarizes the report.

2. Cognitive Model Support

A *cognitive model* describes the cognitive processes and knowledge structures used to form a mental representation of the program under study. Numerous theories have been formulated and empirical studies conducted to explain and document the problem-solving behavior of software engineers engaged in program understanding. von Mayrhauser and Vans surveyed this area in [von Mayrhauser 95] and compared six cognitive models of program understanding.

Rather than impose a process that is not justified by a cognitive model other than that of the environment's developers, the environment should support the diverse cognitive processes and different approaches to program comprehension that the end user prefers. Storey *et al* describe a hierarchy of cognitive design elements to support the construction of a mental model to aid program understanding in software-exploration tools [Storey 97a]. One portion of the hierarchy concerns improving program understanding, such as enhancing bottom-up comprehension by supporting the actions of identifying software artifacts and the relations between them, by browsing code in delocalized plans, and by building abstractions. These actions are in fact composed of one or more canonical reverse-engineering activities that are described in Section 4. For example, the effect of delocalized plans can be reduced through the use of information-analysis techniques, such as slicing, and information-presentation techniques, such as multiple views, supported by the appropriate knowledge-management capabilities.

2.1 Bottom Up

Two common approaches to program understanding often cited in the literature are a functional approach that emphasizes cognition by *what* the system does and a behavioral approach that emphasizes *how* the system works. These two approaches are directly related to the level of domain expertise of the software engineer. The functional approach is *bottom up* and deductive, relying more on the knowledge of the implementation domain to create more abstract concepts that may map to the application domain and the system's functional requirements. The bottom-up approach reconstructs the high-level design of a system, starting with source code, through a series of *chunking* and concept-assignment steps.

2.2 Top Down

The behavioral approach is *top down* and inductive, using a goal-driven method of hypothesis postulation and refinement based on expected artifacts derived from knowledge of the application domain. The top-down approach begins with a pre-existing notion of the functionality

of the system and proceeds to earmark individual components of the system responsible for specific tasks.

2.3 Opportunistic

Both top-down and bottom-up comprehension models have been used in an attempt to define how a software engineer understands a program. However, case studies have shown that, in industry, maintainers of large-scale programs frequently switch between these different models depending on the problem-solving task at hand [von Mayrhauser 92]. This *opportunistic* approach involves creating, verifying, and modifying hypotheses until the entire system can be explained using a consistent set of hypotheses. The opportunistic model describes the maintainer as "... an opportunistic processor capable of exploiting both top-down and bottom-up cues as they become available."

3. Reverse-Engineering Tasks

There are many different reverse-engineering tasks. This section discusses several of the most important: program analysis, plan recognition, concept assignment, redocumentation, and architecture recovery. The first three tasks can be viewed as pattern matching at different levels of abstraction. As illustrated in Figure 1, program analysis is syntactic pattern matching in the programming-language domain, plan recognition is semantic pattern matching in the programming-language domain, and concept assignment is semantic pattern matching in the application (or end-user) domain.

Assigning each task to an abstraction layer can be difficult. One can argue that redocumentation is a form of reverse engineering, or that it is simply restructuring at the same abstraction level. How one interprets each type of reverse engineering depends on several factors, such as which document you read, what you mean by reverse engineering, and what you mean by redocumentation. Many of these arguments are more in the lines of religion rather than practical differences. Suffice it to say that reverse engineering is not an exact science, and neither is its terminology.

Reverse-Engineering Task	Pattern Abstraction Level	Artifacts Manipulated
Program analysis	Programming language, syntactic	Tokens
Plan recognition	Programming language, semantic	Plans
Concept assignment	Application, semantic	Concepts

Figure 1: Pattern Abstraction for Levels of Program Analysis

3.1 Program Analysis

Most commercial systems focus on source-code analysis and simple code restructuring using the most common form of reverse engineering: program analysis. Catalogs such as [Olsem 93, Zvegintzov 94] describe several hundred such packages. Representative types of program analysis include control-flow and data-flow analysis, slicing, and structure charts.

There are many ways of classifying program-analysis techniques. For example, in [Ning 89], Ning identified four levels of abstraction for reverse engineering: implementation, structural,

functional, and domain. The implementation-level view examines individual programming constructs; the program is typically represented as an abstract syntax tree (AST), symbol table, or plain source text. The structural-level view examines the structural relationships among the program constructs; dependencies among program components are explicitly represented. The functional-level view examines the relationships between program structures and their behavior (function); the rationale behind program constructs is also investigated. The domain-level view examines concepts specific to the application domain.

Program-analysis techniques may consider source code in increasingly abstract forms, including raw text, preprocessed text, lexical tokens, syntax trees, annotated abstract syntax trees with symbol tables, control/data flow graphs, program plans, and conceptual models. The more abstract forms entail additional syntactic and semantic analysis that corresponds more to the meaning and behavior of the code and less to the form and structure. Different levels of analysis are necessary for different users and different program-understanding purposes.

3.2 Plan Recognition

Software engineers usually look for code that fits certain patterns. Those patterns that are common and stereotypical are known as *clichés*. Patterns can be structural or behavioral, depending on whether one is searching for code that has a specified syntactic structure, or searching for code components that share specific data-flow, control-flow, or dynamic (program execution-related) relationships.

To locate such patterns, what is needed is a search mechanism that is closer to the mental model of the software engineer than that provided by most program analysis tools that focus on simple statistical and cross-reference queries. This mechanism is called *plan recognition*, which attempts to discover instances of abstract representations of commonly used algorithms and/or data structures in the subject system. Program plans are abstract representations of source code fragments. Comparison methods are used to help recognize instances of programming plans in a subject system. This process involves pattern matching at the programming-language semantic level.

One focus in plan recognition is on identifying similar code fragments. Existing source code is often reused within a system via "cut-and-paste" or "clone-and-own" text operations. This practice saves development time, but leads to problems during maintenance because of the increased code size and the need to propagate changes to every modified copy. Cloned code fragments can be detected using heuristics, since the decision whether two arbitrary programs perform the same function is undecidable. These heuristics are based on the observation that the clones are not arbitrary and will often carry identifiable characteristics (features) of the original fragment. These characteristics are used to compare two code fragments based on similarity measures.

3.3 Concept Assignment

Plan recognition is an improvement over the syntactic pattern matching found in most program-analysis tools. However, when it comes to locating source code fragments of interest to the software engineer (because plans are closer to the programmer's mental model than syntactic entities), the program plans must still be couched in terms of the implementation programming language (or some abstraction thereof). It would be better if program plans represented application-level concepts and were not simply abstracted code fragments.

One approach to this problem is *concept assignment* [Biggerstaff 93]. Concept assignment is the task of discovering individual human-oriented concepts and assigning them to their implementation-oriented counterparts in the subject system. It is related to *teleological maintenance* [Karakostas 90], which attempts to recover information from the subject system based on a specific user model (for example business rules), rather than from the source code. This type of conceptual pattern matching enables the maintainer to search the underlying code base for program fragments that implement a concept from the application. This is advantageous since change requests are usually couched in end-user terminology, not in that of the implementation.

Concept assignment is pattern matching at the end-user application semantic level. It is a process of recognizing concepts within the source code and building an understanding of the program by relating the recognized concepts to portions of the program. Concept recognition is still at the early research stage, in part because automated understanding capability can be quite limited due to difficulties in knowledge acquisition (the identification and specification of plans) and the complexity of the matching process.

3.4 Redocumentation

The lack of detailed, accurate, and up-to-date program documentation is critical for software engineers and technical managers who are responsible for the evolution of existing software systems. Without this documentation, the only reliable and objective information is the source code itself [Fletton 88]. Personnel must spend an inordinate amount of time attempting to create an abstract representation of the system's high-level functionality by exploring its low-level source code. One way of producing accurate documentation for an existing software system is through *redocumentation.*

Redocumentation is one of the oldest forms of reverse engineering [Sneed 84]. It is the process of retroactively providing documentation for an existing software system. If the redocumentation takes the form of modifying commentary within source code, it can be considered a weak form of restructuring. However, it can also be classified as a sub-area of reverse engineering because the reconstructed documentation is typically used to aid program understanding. One can think of it as a transformation from source code to pseudo-code and/or prose, the latter of which is usually considered to be at a higher abstraction level than the former.

The documentation produced is typically in-line text. However, it can take many other forms, including that of linked documentation accessible via hypertext [Tilley 91], cross-reference listings, or graphical views of the software system's artifacts and relationships [Tilley 92]. Some of the newer reverse-engineering environments also support augmenting the source code with multimedia annotations.

3.5 Architecture Recovery

As stated above, documentation has traditionally served an important role in aiding program understanding. However, there are significant differences in documentation needs for software systems of vastly different scales (1,000 lines versus 1,000,000 lines). Most software documentation is "in-the-small," since it typically describes the program at the algorithm and data structure level. For large legacy systems, an understanding of the structural aspects of the system's architecture is more important than any single algorithmic component.

Program understanding is especially problematic for software engineers and technical managers who are responsible for the maintenance of such systems. The documentation that exists for these systems usually describes isolated parts of the system; it does not describe the overall architecture. Moreover, the documentation is often scattered throughout the system and on different media. It is left to maintenance personnel to explore the low-level source code and piece together disparate information to form high-level structural models. Manually creating just one such architectural document is always arduous; creating the necessary documents that describe the architecture from multiple points of view is often impossible. Yet it is exactly this sort of "in-the-large" documentation that is needed to expose the structure of large software systems.

Using reverse engineering to reconstruct the architectural aspects of software may be termed *architecture recovery* [Kazman 97] or *structural redocumentation* [Wong 95]. As a result of this task, the overall gestalt of the subject system can be derived and some of its architectural design information can be recaptured. As with redocumentation, structural redocumentation does not involve physically restructuring the code (although this might be a desirable outcome).

4. Canonical Activities

As stated in Section 1, reverse engineering is the predominant support mechanism used to aid program understanding. It is seen as an activity that does not change the subject system; it is a process of examination, not a process of alteration. It can aid program understanding by directly supporting the essence of program understanding: identifying artifacts, discovering relationships, and generating abstractions. This process depends on several factors, including one's cognitive abilities and preferences, one's familiarity with the application domain, and the set of support facilities provided by the reverse engineering environment.

The artifacts manipulated during reverse engineering can be classified into three categories:

1. data: the factual information used as the basis for study, reasoning, or discussion
2. knowledge: the sum of what is known, which includes data and information such as relationships and rules progressively derived from the data
3. information: contextually and selectively communicated knowledge

The data artifacts are the raw bricks used as building blocks to support program understanding. They form the foundation for the higher level knowledge artifacts. The information artifacts can be created by abstracting up from the data artifacts and matching with expected results from knowledge artifacts.

Based on the description of these artifacts manipulated during the reverse-engineering process, three canonical reverse-engineering activities emerge: data gathering, knowledge management, and information exploration (includes navigation, analysis, and presentation). All tasks carried out by a software engineer during program understanding can be mapped to a composition of one or more of these canonical activities supported by a reverse-engineering environment. The next three sections provide more detailed descriptions of them.

4.1 Data Gathering

To identify the artifacts and relationships of a system and use them to later construct and explore higher level abstractions, raw data about the system must be gathered. Hence, *data gathering* is an essential reverse-engineering activity. It is usually, but not always, the first step. The raw data are used to identify a system's artifacts and relationships; without these data, higher level abstractions cannot be constructed and explored. 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. Be-

cause data represent 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.

Techniques used for data gathering include system examination, document scanning, and experience capture. In addition to using data gathered from traditional sources, such as compiler-based static analysis, it is also possible to integrate alternative sources of data. Examples include 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 nontraditional 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.

Regardless of 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 [Shaw 89].

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.

4.1.1 System Examination

System-examination techniques can be classified as static or dynamic. Static examination focuses primarily on analyzing the program's source code. Dynamic examination focuses primarily on analyzing the executing system.

4.1.1.1 Static Analysis

The predominant technique used for gathering data is *static analysis* by parsing a system's source code to construct abstract syntax trees with a large number of fine-grained syntactic artifacts and dependencies. This type of data gathering is essentially the same as running the front end of a compiler. It requires constructing a scanner and using a valid grammar for the implementation language of the system.

Creating a parser for a modern language like C++ or a legacy language like PL/I is a non-trivial task. 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 [Murphy 96]. Practitioners and researchers alike would benefit greatly if traditional tools, such as compilers, were integrated in newer program understanding tools. This would produce data that are more trusted and accurate.

4.1.1.2 Dynamic Analysis

Dynamic analysis techniques, such as profiling, provide data that can aid the understanding of distributed, real-time, or client-server programs. These types of applications are becoming increasingly predominant, and will soon become legacy systems themselves. Dynamic analysis is also particularly useful for analyzing component-based systems, especially when the components are commercial off-the-shelf (COTS) products.

Components usually do not come with source code, so most of the static program-analysis techniques currently in use are not applicable (with the exception of binary reverse engineering). In the case of component-based systems, dynamic analysis of the running system is a more fruitful endeavor. It can provide more data on the interactions between components in the system, on the types of messages and protocols used, and on the external resources used by the system. All this data is an aid to understanding the overall system.

4.1.2 Document Scanning

Another form of data gathering that does not rely on the programming-language constructs of the source code is *document scanning*. For example, in-line comments[1] 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.

1 Comments written in the same file as the source code.

4.1.3 Experience Capture

Another source of data about software programs are the software engineers responsible for its ongoing evolution. 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. Unfortunately, as discussed in Section 1, this type of *experience capture* is not always possible. See Section 4.2.2 for a further discussion of using this type of gathered data.

4.2 Knowledge Management

As the portion of corporate assets defined as intellectual capital increases, interest in *knowledge-management* techniques increases accordingly [Lang 97]. Knowledge management refers to capturing, organizing, understanding, and extending past experiences, processes, and individual know-how. If managed properly, such artifacts could be shared by all involved in a project, thus serving as an active repository of corporate knowledge. The management of this type of knowledge is valuable in many domains, such as consulting, where large organizations are attempting to make use of pooled knowledge as a strategic advantage.

Knowledge management is equally important in aiding program understanding, where it serves similar purposes. Leveraging corporate knowledge, as described in Section 4.1.3, is one aspect of program understanding that directly benefits from knowledge management. Perhaps one of the most important ways knowledge management techniques aid program understanding is in the creation of domain models. A domain model is a representation that captures the structure and composition of artifacts with a problem area [Tracz 94]. A domain model may be constructed through domain analysis—the process of identifying, organizing, and representing the structure and composition of elements in a domain. Program understanding relies on knowledge-management techniques such as domain modeling to create, represent, and reason about the artifacts and relationships of interest.

This section describes some of the desirable knowledge-management attributes of a reverse-engineering tool. The attributes are organization, evolution, and discovery. Knowledge organization describes the mechanisms used to structure the gathered data into a form more amenable to representing the application domain and supporting the desired operations on the data. Knowledge discovery describes the techniques used to support information exploration. Knowledge evolution is concerned with updating the knowledge about the subject system during the reverse-engineering process (for example, extending the schema without having to recreate the database).

4.2.1 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 requirement—the need to organize data in some well-defined and rigorous manner—led to the development of data models [Borkin 80]. A data model enables us to understand the essential properties and relationships between artifacts in

a system. Without a model, raw data are almost impossible to understand. *Knowledge management* techniques are used to create, represent, and reason about data models and to structure the data into a conceptual model of the application domain.

A data model captures the static and dynamic properties of an application needed to support the desired data-related processes. An application can be characterized by static properties (such as objects, attributes, and relationships among objects), dynamic properties (such as operations on objects, operation properties, and relationships among operations), and integrity constraints over objects and operations. The result of data modeling is a representation that has two components: static properties that are defined in a schema and dynamic properties that are defined as specifications for transactions, queries, and reports. A schema consists of a definition of all application object types, including their attributes, relationships, and static constraints. Corresponding to the schema is a data repository called a database, an instance of the schema. A data model provides a formal basis for tools and techniques used to support data modeling.

The three best-known classical data models are the hierarchical data model, the network data model, and the relational data model [Ullman 80]. The hierarchical data model is a direct extension of a primitive file-based data model; data are organized into simple tree structures. The network model is a superset of the hierarchical model; the objects need not be tree structured. The relational model is quite different from the hierarchical or network model; it is based on the mathematical concept of a relation (a set of *n*–tuples), and organizes data as a collection of tables. All three classical data models are instances of the record-based logical data model [Korth 86].

Although well suited to a computer environment, record-oriented data models are often semantically inadequate for modeling the application environment. They are highly machine oriented and organized for efficiency of storage and retrieval operations; ease of use for the non-programmer is of secondary importance. Typically, only two levels of abstraction are provided: the database schema and the actual collection of records. There are no provisions to extend the levels to a more general hierarchy of types, meta-types, and instances, even though this extension would increase the model's expressive power and provide a mechanism which supports the reuse of common properties. The hierarchical and network models also do not support semantic relativism, which is the ability when modeling a system to view the elements and concepts representing it from different perspectives depending on the application. In particular, the concepts of entity, relationship, and attribute should be interchangeable. For these reasons, the classical data models are also known as syntactic data models.

The lack of abstraction mechanisms provided by the classical data models is particularly troublesome from a program understanding point of view. Abstraction is a fundamental conceptual tool used for organizing information. It plays a key role in managing one of the fundamental problems with large-scale systems—coping with complexity [Brooks 87]. When modeling such systems, the number of objects and relations in the knowledge base can grow

very large. Like a large software system, a large knowledge base needs organizational principles to be understandable. Without these principles, a knowledge base can be as unmanageable as a program written in a language that has no abstraction facilities.

Abstraction is the selective emphasis on detail: specific details are suppressed and those pertinent to the problem at hand are emphasized. Abstraction mechanisms serve as organizational axes for structuring the knowledge base. They focus on high-level aspects of an entity while concealing details. Three of the most common abstraction mechanisms used are classification, aggregation, and generalization [Sowa 88]:

- **Classification** is a form of abstraction in which an object type is defined as a set of instances. It captures common characteristics shared by a collection of objects, resulting in a generic object which captures the essential similarity among its constituents. An *instance-of* relationship is established between an object type in the schema and its instance in the knowledge base.
- **Aggregation** is a form of abstraction in which a relationship between objects is considered as a higher level aggregate object. When considering the aggregate, specific details of the constituent objects are suppressed. A *part-of* relationship is established between the component objects and the aggregate object.
- **Generalization** is a form of abstraction in which similar objects are related to a higher level generic object. The constituent objects are considered specializations of the generic object. An *is-a* relationship is established between the specialized objects and the generic object.

There have been two basic approaches to addressing some of the deficiencies in the classical data models to "capture more of the semantics of an application [Codd 79]." Attempts have been made to extend the classical models by building higher level conceptual models on top of them, and new more powerful semantic data models have also been developed to capture database concepts at a more user-oriented level. Semantic data models, starting with Abrial's semantic model [Abrial 74] and Chen's entity-relationship model [Chen 76], combined simple knowledge-representation techniques, often borrowed from semantic networks [Findler 79], with database technology. Semantic data models represent a shift in database research away from the traditional record-oriented model towards models that support more human-oriented semantic constructs. This shift is very similar to the goals in programming language research focusing on abstraction mechanisms for software development and artificial-intelligence research into knowledge representation based on network representation schemes [Gilbert 90]. *Conceptual modeling* was introduced as a term reflecting this broader perspective [Brodie 84].

Conceptual modeling is the activity of formally describing aspects of some information space for the purpose of understanding and communication. Such descriptions are often referred to as conceptual schemata. A conceptual model and a conceptual schema are analogous to a data model and a database schema, respectively. One can think of data models as special conceptual models where the intended subject matter consists of data structures and associated operations. Classical data models, grounded on mathematical and computer science concepts,

such as relations and records, offer little to aid database designers and users in interpreting the contents of a database.

Semantic data modeling shares purposes with conceptual modeling. However, semantic data modeling introduces assumptions about the way conceptual schemata will be realized on a physical machine (the "data-modeling" dimension). Thus, semantic data modeling can be seen as a more constrained activity than conceptual modeling, leading to simpler notations, but also ones that are closer to the implementation.

The fundamental characteristic of conceptual modeling is that it is closer to the human conceptualization of a problem domain than to a computer representation of the problem domain [Kristensen 94]. The emphasis is on *knowledge organization* (modeling entities and their semantic relationships), rather than on data organization. The descriptions that arise from conceptual-modeling activities are intended to be used by humans—not machines. Concepts in a conceptual model are indexed by their semantic content. This differs from other data models, such as relational, where the indexing scheme is geared more towards optimal storage and information retrieval from the implementation perspective. This is one of the main reasons that conceptual modeling is eminently suited to program understanding: the focus on the end user is paramount.

The sometimes conflicting requirements for organizing the three different categories of artifacts (data, knowledge, and information) suggest that a single technique for representing them may not always be suitable. In its place, a layered approach may be used: for each type of artifact manipulated during the reverse-engineering process, a different model may be used [Tilley 95b]. The advantage of such an approach is that different technologies may be used to their strengths, while avoiding their weaknesses. For example, a relational model may be used for physical storage of data artifacts, a conceptual model may be used for representing domain-level knowledge, and a semantic network model may be used for interactive discovery.

4.2.2 Discovery

Once knowledge about the problem domain has been organized, one of its primary uses is to aid others in understanding aspects of the problem domain. This *discovery* aspect of knowledge management is directly related to one of the data-gathering techniques described in Section 4.1.3: leveraging corporate knowledge through experience capture. Experience capture is accomplished during information exploration, the canonical reverse-engineering activity discussed in Section 4.3.

Knowledge discovery can best be accomplished by providing multiple perspectives (analogous to database views) on the underlying artifacts. Using a common web[1] metaphor, the

1 The term web refers to a structured information space composed of nodes representing artifacts from the application domain and links representing relationships between the nodes. The web that is part of the Internet is sometimes referred to as "The Web."

software engineer navigates through the hyperspace that represents the information related to the subject system, analyzes this information with respect to domain-specific evaluation criteria, and uses various presentation mechanisms to clarify the resultant information. A general-purpose semantic network, represented as an attributed graph, is well suited to representing such structured sets of artifacts [Rohrich 87].

In its most basic form, a semantic network represents knowledge in terms of a collection of objects (representing concepts) and binary associations (representing binary relations over these concepts). According to this view, a knowledge base is a collection of objects and relations defined over them [Mylopoulos 84]. The semantics of the model are a careful definition of the meaning and usage of the nodes and arcs. Modifications to the knowledge base occur through the insertion or deletion of objects and the manipulation of relations.

The use of a network model has at least three advantages related to navigating, structuring, and visualizing the knowledge base. The first advantage is that the network structures that encode information may themselves serve as a guide for information retrieval [Hendrix 79]. The association between artifacts defines implicit access paths. Using this model, the information space is indexed by neighborhoods, while artifacts are retrieved through navigation guided by spatial and visual proximity cues.

The second advantage is the use of the organizational principles described in the previous section to structure the knowledge base. Such abstraction mechanisms capture the natural structure of the artifacts in the system, their properties, and the relationships among them. They can also be used recursively to construct abstraction hierarchies. These structuring aids can be represented in the semantic network by typing both the nodes and the arcs.

The third advantage is that network representation schemes lend themselves to a graphical notation that can be used to depict knowledge bases and increase their understandability. Most humans visualize structure graphically. For examples, designers often describe system architecture using block diagrams of the major system components and labels that refer to their major functions. Modern interactive systems with graphical display capabilities facilitate the direct manipulation, processing, and presentation of information in graphical form.

Without a web metaphor, a system's knowledge discovery mechanisms should permit the analysis of artifacts and relationships of interest. This means that there should be support for standard database queries. The raw data should be available, as should the conceptual constructs created during knowledge-organization activities. This information is needed so that the software engineer can discover facts about the system from multiple perspectives, for example from the implementation perspective and from the end-user perspective.

4.2.3 Evolution

As part of knowledge organization, the construction of the domain model about the subject system (or about the subject system's application domain) 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.

Iterative domain modeling is one form of knowledge evolution. A software engineer can use tools that support iterative domain modeling to recognize standard components of a system automatically and use these components to populate the domain model. This type of top-down construction can be used as a guide during program understanding.

The software engineer can also use semi-automatic or manual techniques to classify nonstandard components and use this information to extend the domain model. In database terminology, this is known as dynamic schema evolution. It is another form of knowledge evolution that can be used during exploratory program understanding, when hypotheses about the subject system are being tested and theories revised.

4.3 Information Exploration

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

Exploration is a composite activity that includes navigation, analysis, and presentation. Information exploration makes use of the knowledge-discovery structures discussed in Section 4.2.2. Using the same web metaphor, the software engineer navigates through the structured information space that represents the information related to the subject system. As part of the exploration, the information is analyzed and filtered with respect to domain-specific criteria. Various presentation mechanisms are used to clarify the resultant information.

4.3.1 Navigation

Large software systems, like other complex systems, are nonlinear and may be viewed as consisting of an interwoven and multidimensional web of information artifacts [Maurer 92]. 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. Hypermedia-based *information navigation* allows software engineers to traverse this "information web" as part of their exploratory understanding activities. The information-navigation activity can itself be subdivided into selection, editing, and traversal.

4.3.1.1 Selection

Selection is one of the most important of all canonical activities because it is related to the essence of program understanding—identifying artifacts and understanding their relationships. Software engineers must first find the relevant code before they can transform it. 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. This is essentially a pattern-matching activity at various abstraction levels. This pattern recognition is accomplished either mentally by the software engineer or mechanically by the reverse-engineering environment. Artifacts are segmented into features, patterns of which 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.

Since searching for code is an extremely common reverse-engineering activity, sophisticated selection tools can greatly aid the process. Artifacts can be selected according to various criteria, including visual and spatial cues, attributes (such as names), and structural properties (for composite artifacts). The type of pattern recognition provided can range from recognition of simple regular expressions, such as that provided by the UNIX grep tool, to more advanced capabilities such as plan recognition. Some query mechanisms enable users to specify attribute patterns that are used to identify artifacts in the database that satisfy the search criteria. A more abstract mechanism is the use of powerful query and analysis languages that do not involve procedural code, such as the Source Code Algebra (SCA) query formalism [Paul 95]. The plan-recognition and concept-assignment tasks described in Section 3 are in essence advanced selection activities.

To aid information selection, it is beneficial to be able to augment the operations built into the reverse-engineering support mechanism with 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. Program understanding can be improved by leveraging external tools that provide advanced searching techniques and having the results of their searches made available to the user and the environment.

4.3.1.2 Editing

Editing is an activity that can alter the knowledge organization structure, sometimes as a byproduct of information navigation. It can involve creating new artifacts, deleting existing ones, or changing an artifact's attributes. For example, through editing activities, a user may create user-specified subsystem constructs that are logical (but not physical) representations

of the system. Information editing is therefore the activity that supports the evolution of the knowledge base as discussed in Section 4.2.3.

4.3.1.3 Traversal

Traversal is the action of moving from one artifact to another in the information space. For example, following links in the web that represent relations such as "calls" involves traversal. Unfortunately, as the size of this information space grows, the well-known "lost in hyperspace" syndrome may limit navigational efficiency [Marchionini 88].

Disorientation has been attributed to the tangle of links in the web [Nielsen 90a]. The proliferation of links is often due to the weak link discipline enforced by a system using a simple node/link mechanism, allowing unrestricted linking among arbitrary objects [Nanard 91]. Such linking is very powerful, but potentially disorienting [Broady 93]. The same freedom that provides hypertext's flexible structure and browsing capabilities may also be the direct cause of one of its greatest problems [Botagofo 91]. During knowledge discovery, disorientation may occur when browsing. During knowledge evolution, the lack of design principles when creating associative links does not foster the creation of a consistent conceptual model [Hara 91].

Clearly, reducing disorientation is a key capability that a support mechanism should address. Some of the solutions that have been proposed to the classical problem of user disorientation within a large information space include maps, multiple windows, history lists, and tour/path mechanisms [Nielsen 90b]. Unfortunately, many of these methods are not sufficiently scalable. A more successful approach is the use of composite nodes; they reduce web complexity and simplify its structure by clustering nodes together to form more abstract, aggregate objects [Casanova 91]. Composite nodes interact with sets of nodes as unique entities, separate from their components.

4.3.2 Analysis

The critical step in deriving abstractions from the raw data to foster understanding is *analysis*. Software engineers use the resultant information to explore the system further. There are many forms of analysis; the complete list is not enumerated in this report. The level of analysis is directly related to the type of canonical artifact being manipulated. The degree of manipulation is governed in part by the level of automation provided by the reverse engineering environment.

4.3.2.1 Types

There are a great many *types* of analysis possible; this sub-category is both very broad and very deep. For example, slicing is an analysis technique that 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.

There are two forms of analysis that are directly related to two forms of data gathering: static analysis and dynamic analysis. Most reverse-engineering tools provide a variety of static-analysis capabilities (for example, def/use analysis for data types and variable instances). There is less support for dynamic analysis. This may be due to environmental requirements; by necessity, dynamic analysis requires the program to execute in a real or simulated manner. However, some types of analysis, such as slicing, come in both static and dynamic varieties. Other common forms of analysis include the traditional complexity metrics and perhaps the most important—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, enabling the software engineer to function at the application-domain level rather than the implementation-domain level.

4.3.2.2 Levels

The type of analysis supported is closely related to the abstraction *level* provided by the pattern-recognition capabilities of the tool. Program-understanding techniques can consider source code in increasingly abstract forms: raw text, preprocessed text, lexical tokens, syntax trees, control and data-flow graphs, program plans, architectural descriptions, and conceptual models. The more abstract forms entail additional syntactic and semantic analysis that corresponds more to the meaning and behavior of the code and less to its form and structure. Different levels of analysis are necessary for different users and different reverse-engineering activities.

Analyzing the structure of the information web can provide useful insight. Various metrics and measures can be used to guide the creation of new artifacts in the information space by editing the information web as part of the knowledge-evolution activity described in Sections 4.2.3 and 4.3.1.2. The environment should support the integration of external analysis packages that implement domain-specific metrics; this is a related quality attributed discussed in Section 5.2.

4.3.2.3 Automation

It is important to manage the tradeoff between the analyses that are handled automatically by a reverse-engineering environment and the functions that enable the tools in the environment to accept human input and guidance. Issues include how to best balance between automatic, semi-automatic, and manual analysis, 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. More analysis automation is likely to occur as the problem characteristics become better understood. This is already happening with regard to the so-called "Year 2000" problem [Smith 97], where the types of analysis are well defined and the types of remediation are limited.

Analysis derives and extracts information that is not explicitly available and generates insightful views that can aid the understanding of the underlying system. 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, thereby extending the level of automation on an as-needed basis.

4.3.3 Presentation

Visual metaphors are commonly used to communicate information. Reverse-engineering environments that provide flexible *presentation* mechanisms that capture such metaphors can aid program understanding. Most reverse-engineering systems provide the user with fixed presentation options, such as cross-reference graphs or module-structure charts, that summarize the results of analyses such as those described in Section 4.3.2. Even though the developers of the environment might consider fixed options to be adequate, there are always users who want something else. Ideally, it should be possible to create multiple, perhaps orthogonal, structures (as described in Section 4.2.1) and to view them using a variety of mechanisms.

4.3.3.1 Multiple Views

It is difficult to convey and communicate the wealth of information generated as a result of reverse engineering. This problem is exacerbated by the necessary coexistence of spatial and visual data [Müller 92]. Theories of cognition suggest that imagery involves both descriptive and depictive information [Kosslyn 80]. For program understanding, both spatial and visual information seem to play key roles in forming mental models of structure.

The spatial component constitutes information about the relative positions of the artifacts in a neighborhood. It provides low-level, detailed information concerning the immediate neighborhood of the artifact in a graphical representation that facilitates the systematic exploration of the structure. The visual component preserves information about how a neighborhood[1] looks (e.g., size, shape, or density). It provides a high-level view of the neighborhood—the essence of the entire image. Visual graph representations aim to exploit the ability of the human visual system to recognize and appreciate patterns and motifs such as central, fringe, or isolated components.

1 A *neighborhood* in this context is a visual representation of the current perspective of the underlying software system, using the web metaphor described in Section 4.2.2.

The way information is presented should not be fixed by the environment. It should be possible to create multiple, perhaps orthogonal, structures and view them using a variety of mechanisms, such as using different graph layouts provided by external toolkits. These views are a crucial component of the knowledge-discovery activity described in Section 4.2.2. The notion of multiple views is not new; they have been employed in the database domain for some time.

4.3.3.2 Visualization Techniques

Graph-layout theory has already proven effective in aiding program understanding; graphical representations of source code proliferate in current reverse-engineering systems. Refinements to this traditional area also show promise (for example, so-called "fish-eye" views that emphasize selected focal points while retaining relative location information). Recent work has explored the coexistence of multiple views (see also Section 4.3.3.1) of the underlying data using a variant of fish-eye views and traditional hierarchical structure diagrams [Storey 97b]. More advanced *visualization techniques* using three-dimensional data imaging, virtual reality "code walk-through," and user-defined views are in the experimental stage. One or more of these techniques can provide new insights during program understanding.

Presentation of analysis results has traditionally taken the form of charts, tables, or graphs. The 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.

4.3.3.3 User Interface

Presentation integration can occur at different levels, including the window manager, the toolkit used to build applications, and the toolkit's "look and feel" [Wasserman 89]. The standardization provided by presentation integration lessens the "cognitive surprise" experienced by users when switching between tools. However, what is really needed is a way for the user to specify the common look and feel of the applications they are interested in, or of tools that are part of an application [Klefstad 88]. In other words, users need to be able to impose their own personal taste on the *user interface*. This refinement of presentation integration moves the onus—and the opportunity—for reducing cognitive overhead induced by the user interface from the tool builder to the tool user.

The goal of environmental customizability includes modification of the system's interface components such as buttons, dialogs, menus, scrollbars, and of the integration of external tools that present the information in different ways. Since the user interface is a crucial part of the infrastructure of many software environments [Young 88], and since personal preferences for things such as menu structure, mouse action, and system functionality differ so much from person to person (and from domain to domain), it is unlikely that any single choice made by the tool builder will suit all users. Most popular PC software applications now provide some level of user interface customizability.

5. Quality Attributes

A *quality attribute* is defined by Barbacci *et al* as a system requirement that is essentially non-functional in nature [Barbacci 95]. Examples of quality attributes include dependability, extensibility, and usability. This section briefly describes selected quality attributes that are of particular importance to most support mechanisms: applicability, extensibility, and scalability. This set of quality attributes is by no means exhaustive. Other attributes, such as usability and deployability, may be equally important in certain circumstances. However, they are more general attributes that are applicable to almost any software system. As such, they should be considered when investigating a particular reverse-engineering environment, but we will not elaborate on them here.

5.1 Applicability

The *applicability* of the support mechanism refers to a particular domain. While a domain may be generally defined as a problem area, domain is an overburdened term. It can refer to both the application domain and the implementation domains. It is naturally desirable to make the support mechanism as flexible as possible for use in many different domains.

One way to maximize the support mechanism's usefulness is to make it domain specific. By doing so, one can provide users with a system tailored to a certain task and exploit any features that make performing this task easier. However, this approach limits the system's usefulness to a particular domain. Using the same system on a different task, even one that is similar, may well be impossible. For example, many current reverse-engineering environments support only relatively small programs. Others support just one programming language (or a subset of it), usually because their parsing system, database, and support environments are tightly coupled. This approach limits the application domain to small, "pure" programs rarely found in practice. One must take a pragmatic point of view: if the support mechanism does not work on real-world software systems, with all their "features," then it will not make an impact on existing systems.

An alternative to making the support mechanism powerful by making it domain specific is to make it domain retargetable [Tilley 95a]. One would like to make the approach as flexible as possible—a subtle distinction from general. Software can be considered general if it can be used without change; it is flexible if it can be easily adapted to be used in a variety of situations [Parnas 79]. General solutions often suffer from poor performance or lack of features that limit their usefulness. Flexible solutions may be tailored by the user to fully exploit aspects of the problem that make its solution easier.

5.1.1 Application Domain

The *application domain* of a reverse-engineering tool is one of its strongest differentiators. Some tools provide a programmable environment with which one constructs more specialized tools. Others are targeted to narrower domains, such as redocumenting C programs.

Subject system characteristics have considerable impact on the applicability of reverse-engineering environments. Different approaches to understanding are used in different application domains. For example, the reconstruction of the schema for database systems, the monitoring of task allocation in distributed systems, and the dynamic analysis of task rendezvous in real-time systems all have different requirements. Different application domains and example representative functional requirements of the support mechanism are illustrated in Figure 2.

Application Domain	Representative Support Mechanism Requirements
Database system	schema reconstruction, data reverse engineering
Distributed system	task allocation, inter-process communication
Real-time system	dynamic analysis, timing constraint maintenance

Figure 2: Application Domains and Support Mechanism Requirements

5.1.2 Implementation Domain

The subject system's *implementation domain* can also play an important role in the selection of the appropriate support mechanism. Issues to consider include the language dialect or variant, the robustness of the parsing mechanism used (e.g., whether or not it supports implementation-language extensions, syntactically incorrect constructs, or incomplete code fragments), and whether or not mixed-mode source code is supported. For example, older C programs may be written in K&R style, many FORTRAN programs make use of machine- or compiler-specific extensions to increase the performance of the compiled code, and COBOL programs may include database preprocessor directives. Some of these considerations are shown in Figure 3.

Implementation Domain	Representative Support Mechanism Requirements
C	K&R, ANSI, dialects, C++
FORTRAN	FORTRAN 66 anachronisms, FORTRAN 77 extensions, FORTRAN 90 constructs
COBOL	CICS, OO-COBOL additions, dialects

Figure 3: Implementation Domains and Support Mechanism Requirements

5.2 Extensibility

Most reverse-engineering environments provide the user with a fixed set of capabilities. While this set might be considered large by the system's producers, there will always be users who want something else. One cannot predict which aspects of a system are important for all users, and how these aspects should be documented, represented, and presented to the user. Hence, the *extensibility* of the environment is an important quality attribute.

5.2.1 Integration Mechanisms

There are many tradeoffs between open and closed systems. An open system provides composable operations and mechanisms for user-defined extensions. A closed system provides a "large" set of built-in facilities, but no way of extending the set. Instead of a closed architecture, a reverse-engineering environment should provide *integration mechanisms* through which users can extend the system's functionality.

There are two basic approaches to constructing extensible integrated applications from a set of tools: tool integration and tool composition [Arora 93]. In tool integration, each tool must be aware of the larger environment, and the inter-tool interactions are coded in the tools themselves. This works for tightly integrated environments, but it is very hard to achieve in a loosely coupled environment. In tool composition, tool interaction logic resides outside of the tools. Each tool presents a standard, well-known interface to the outside world, and knows nothing about its environment; the environment contains all the inter-tool coordination logic.

The method with which one interacts with the tool can have a great impact on its effectiveness, applicability, and ease of use. Some tools are meant to be controlled by other applications (for example, an embedded data-analysis component). Others are meant to be used as the primary interface to the underlying information space (for example, a Web-based navigation engine). The choice of a command-line interface versus a graphical user interface, while sometimes viewed as a secondary concern, can often be a deciding factor in a program-

understanding tool. This is due in part to the significant role visual information can play during pattern recognition, as outlined in Section 4.3.1.1.

The need to inter-operate with other tools is essential for reverse-engineering environments. One common class of applications that they often work with are computer-aided software engineering (CASE) tools. Usually these CASE tools support some type of design or modeling standard. The support (or lack thereof) of such standards in a reverse-engineering environment can play a major role in certain circumstances.

5.2.2 End-User Programmability

It has been repeatedly shown that no matter how much designers and programmers try to anticipate and provide for users' needs, the effort will always fall short. It is extremely difficult to predict all the ways in which a system will be used. Customizations, extensions, and new applications inevitably become necessary.

Some leading-edge reverse-engineering systems provide full-fledged programming languages that can be used to encode analysis methods. The ability to use *end-user programmability* in reverse engineering to develop analysis techniques for specific tasks enhances the analysis power of the support mechanism. For example, domain-specific scripts can increase the likelihood that analyses will better apply to unique software systems.

From an end-user perspective, the reverse-engineering environment should manage tool composition to facilitate the introduction of new tools into the system. This would enable end users to provide new parsing engines, for example, if the application implementation language was not directly supported by the reverse-engineering environment. For tasks such as transformation and pattern matching, the addition of a programmable interface can transform an environment from a collection of individual support mechanisms into an integrated exploratory workbench.

5.2.3 Automatability

While creating the semantic abstractions that facilitate program understanding, it should be possible to include human input and expertise in the decision making. There is a tradeoff between what can be automated and what should or must be left to humans; the best solution lies in a combination of the two. Hence, the construction of abstract representations manually, semi-automatically, or automatically (where applicable) should be possible. Through user control, the reverse-engineering process can be based on diverse criteria such as business policies, tax laws, or other semantic information not directly accessible from the gathered data. This quality attribute is related to the canonical activity of analysis automation described in Section 4.3.2.3.

5.3 Scalability

Ideally, any reverse-engineering environment should be applicable to large systems. As mentioned in Section 4.2.1, the volume of data produced during the reverse engineering of a large-scale software system is considerable. Therefore, *scalability* is an important quality attribute.

Large data sets and information complexity require scalable knowledge bases that use fundamentally different approaches to repository technology than is used in other application domains. Knowledge organization, search strategies, and human-computer interfaces that work on systems "in-the-small" often do not scale up. To gain useful knowledge, one must effectively summarize and abstract the information.

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. A scalable knowledge base improves the understanding of large software systems.

6. Miscellaneous Characteristics

There are a wide variety of system characteristics that do not properly fit into any of the other categories discussed above. However, when it comes to investigating a particular reverse-engineering environment, they can be equally important. As with the quality attributes described in Section 5, the set of miscellaneous characteristics included in this section is representative, not exhaustive.

6.1 Computing Platform

The particular hardware and software platform on which the reverse-engineering environment will run can have an immediate impact on whether or not it is appropriate for a particular program-understanding task. For example, if dynamic analysis of the running program is needed, then the computing platform that hosts the reverse-engineering environment must be the same, or there must be a mechanism for gathering the data and incorporating it into the analysis engine(s) of the environment.

6.2 Ancillary Requirements

The ancillary requirements of the reverse-engineering environment should not be such that the requirements preclude the system's use. For example, the environment may require the user to have a valid license for a particular database system that is used by the underlying knowledge-management system. The requirements can also affect the computing platform, which in turn affects the selection of the environment.

6.3 Cost

Perhaps one of the most rudimentary miscellaneous characteristics of any reverse-engineering environment is its cost. The "cost" can be measured in various ways, not just initial purchase price. There is no doubt that the pricing structure of some environments precludes their use by all but very large organizations. But the cost of maintaining the environment (and its ancillary requirements) can outweigh the initial purchase price. This is especially true if personnel must be specially trained in its use, and if someone must be designated the equivalent of "gatekeeper" for the environment (for example, for maintaining the system-wide schema used for knowledge management by several teams of people).

There is also the cost of adopting or deploying the tool. This is a characteristic that might also be considered a quality attribute, because adoption and deployment have proven to be such stumbling blocks for many tools over the years. Included in this cost is the time required to train users.

7. Summary

Program understanding is critical to our ability to modernize legacy systems. Operational for many years, legacy systems embody substantial corporate knowledge, including requirements, design decisions, and business rules. As its software ages, the task of maintaining a legacy system becomes more complex and more expensive. Nevertheless, because most successful legacy systems play critical roles in the day-to-day business of an enterprise, the system cannot simply be discarded and replaced; it must be maintained and enhanced. This cannot be accomplished without a sufficient understanding of the program's artifacts and relationships.

There now exists a wide variety of tools and techniques that support program understanding. The most prevalent is computer-aided reverse engineering. This report presented a framework for classifying the features of reverse-engineering environments. The core of the framework is a descriptive model that categorizes the important support mechanisms and features provided by the environments that aid program understanding. The descriptive model is summarized in tabular format in Figure 4. The goals for creating this framework include helping potential users of such environments to independently evaluate claims, assess the applicability of certain approaches to their own software evolution challenges, and facilitate the comparison of similar approaches to the same problem.

This work on reverse-engineering environments described in this report is part of an ongoing research project studying legacy system reengineering, in which program understanding and reverse engineering play a significant role. The conceptual framework described in this report will evolve through peer review, usage experience, and completeness verification. One of the next steps in its evolution is to populate the framework with representative reverse-engineering environments (both research oriented and commercial offerings) and perform experiments to flesh-out the basis of its descriptive model (the canonical activities of reverse engineering and the quality attributes). A preliminary use of the framework in the investigation of a representative commercial offering is reported in [Tilley 97].

Cognitive Model Support	Top-Down		
	Bottom-Up		
	Opportunistic		
Reverse Engineering Tasks	Program Analysis		
	Plan Recognition		
	Concept Assignment		
	Redocumentation		
	Architecture Recovery		
Canonical Activities	Data Gathering	System Examination	Static
			Dynamic
			Mixed
		Document Scanning	
		Experience Capture	
	Knowledge Management	Organization	
		Discovery	
		Evolution	
	Information Exploration	Navigation	Selection
			Editing
			Traversal
		Analysis	Types
			Levels
			Automation
		Presentation	Multiple Views
			Visualization Techniques
			User Interface
Quality Attributes	Applicability	Application Domain	
		Implementation Domain	
	Extensibility	Integration Mechanisms	
		End-User Programmability	
		Automatability	
	Scalability		
Miscellaneous Characteristics	Computing Platform		
	Ancillary Requirements		
	Cost		

Figure 4: The Descriptive Model

Acknowledgements

As described in Section 1.2, Dennis Smith of the Software Engineering Institute and Santanu Paul of IBM Research contributed to earlier versions of this framework. Dennis Smith and Steve Woods of the Software Engineering Institute provided valuable comments on early drafts of this report.

References

[Abrial 74] Abrial, J. R. "Data Semantics." *Data Management Systems* (Klimbie and Koffman, editors). North-Holland, 1974.

[Arora 93] Arora, Adarsh K.; Hurst, David W.; and Ferrans, James C. "Building Diverse Environments with PCTE Workbench," 543-560. *Proceedings of PCTE '93*, 1993. Paris, France, November 1993. Syntagma Systems Literature, 1993.

[Barbacci 95] Barbacci, M. R.; Klein, M. H.; Longstaff, T. H.; and Weinstock, C. B. *Quality Attributes* (CMU/SEI-95-TR-021, ADA 307888). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1995.

[Biggerstaff 93] Biggerstaff, Ted G.; B. G. Mitbander, B. and Webster, D. "The Concept Assignment Problem in Program Understanding," 27-43. *Proceedings of the 1993 Working Conference on Reverse Engineering (WCRE '93).* Baltimore, Maryland, May 21-23, 1993. IEEE Computer Society Press, 1993.

[Borkin 80] Borkin, Sheldon. *Data Models: A Semantic Approach for Database Systems*. Boston, MA: The MIT Press, 1980.

[Botagofo 91] Botagofo, Rodrigo A. and Shneiderman, Ben. "Identifying Aggregates in Hypertext Structures," 63-74. *Proceedings of Hypertext '91*. San Antonio, Texas, December 15-18, 1991. ACM Press, 1991.

[Broady 93] Broady, Donald; Haitto, Hasse; Lidbaum, Peter; and Tobiasson, Magnus. *DARC: Document Archive Controller* (TRITA-NA-P9306). Sweden: IPLab/NADA, Royal Institute of Technology, March 1993.

[Brodie 84] Brodie, Michael L.; Mylopoulos, John; and Schmidt, Joachim W. (editors). *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages.* New York, NY: Springer-Verlag, 1984.

[Brooks 87] Brooks, Frederick P. Jr. "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer 20*, 4 (April 1987): 10-19.

[Casanova 91] Casanova, Marco A.; Tucherman, Luiz; Lima, Maria Julia D.; Netto, Jose L. Rangel; Rodriguez, Noemi; and Soares, Lui F. G. "The Nested Context Model for Hyperdocuments," 193-201. *Proceedings of Hypertext '91*. San Antonio, Texas; December 15-18, 1991. ACM Press, 1991.

[Chen 76] Chen, Peter. "The Entity-Relationship Model: Towards a Unified View of Data," *ACM Transactions on Database Systems 1*, 1 (1976): .

[Codd 79] Codd, E. F. "Extending the Database Relational Model to Capture More Meaning," *ACM Transactions on Database Systems 4*, 4 (December 1979).

[Findler 79] Findler, Nicholas V. "A Heuristic Information Retrieval System Based on Associative Networks," 305-326. *Associative Networks (Representation and Use of Knowledge by Computers)*. New York, NY: Academic Press, 1979.

[Fletton 88] Fletton, Nigel T. and Munro, Malcolm. "Redocumenting Software Systems Using Hypertext Technology, " 54-59. *Proceedings of the 1988 Conference on Software Maintenance (CSM '88)*. Phoenix, Arizona, October 24-27, 1988. IEEE Computer Society Press, 1988.

[Gilbert 90] Gilbert, Jonathan Paul. *PolyView: An Object-Oriented Data Model for Supporting Multiple User Views* (Ph.D. thesis). Irvine, CA: Department of Information and Computer Science, University of California at Irvine, 1990.

[Hara 91] Hara, Yoshinori; Keller, Arthur M.; and Wiederhold, Gio. "Implementing Hypertext Database Relations through Aggregations and Exceptions," 75-90. *Proceedings of Hypertext '91*. San Antonio, Texas; December 15-18, 1991. New York, NY: ACM Press, 1991.

[Hendrix 79] Hendrix, Gary G. "Encoding Knowledge in Partitioned Networks," 51-92. *Associative Networks (Representation and Use of Knowledge by Computers)* (Nicholas V. Findler, editor). New York, NY: Academic Press, 1979.

[Karakostas 90] Karakostas, V. "Modelling and Maintenance Software Systems at the Teleological Level." *Journal of Software Maintenance: Research and Practice* 2 (1990): 47-59.

[Kazman 97] Kazman, Rick and Carrière, S. Jeromy. *Playing Detective: Reconstructing Software Architecture from Available Evidence* (CMU/SEI-97-TR-010, ADA 315653). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.

[Klefstad 88] Klefstad, Raymond O. *Maintaining a Uniform User Interface for an Ada Programming Environment* (Ph.D. thesis). Irvine, CA: Department of Information and Computer Science, University of California, Irvine, 1988.

[Korth 86] Korth, Henry F. and Silberschatz, Abraham. *Database System Concepts.* New York, NY: McGraw-Hill, 1986.

[Kosslyn 80] Kosslyn, S. *Image and Mind.* Cambridge, MA: Harvard University Press, 1980.

[Kristensen 94] Kristensen, Bent Bruun; and Österbye, Kasper. "Conceptual Modeling and Programming Languages." *ACM SIGPLAN 29*, 9 (September 1994): 81.

[Lang 97] Lang, S. and von Mayrhauser, A. "Towards a Systematic Analysis of Program Comprehension Strategies for Legacy Software." *Workshop on Migration Strategies for Legacy Systems*, held in conjunction with the *1997 International Conference on Software Engineering (ICSE-18).* Boston, MA, May 17, 1997. Los Alamitos, CA: IEEE Computer Society Press, 1997.

[Marchionini 88] Marchionini, G. and Shneiderman, B. "Finding Facts and Browsing Knowledge in Hypertext Systems," *Computer 21* (1988):70-80.

[Maurer 92] Maurer, Hermann. *Why Hypermedia Systems Are Important* (Technical Report 331). Graz, Austria: Institutes for Information Processing (IIG), Graz University of Technology, 1992.

[Murphy 96] Murphy, Gail C; Notkin, David; and Lan, E. S. C. "An Empirical Study of Static Call Graph Extractors." *The 18th International Conference on Software Engineering (ICSE-18).* Berlin, Germany, March 29-31, 1996. Los Alamitos, CA: IEEE Computer Society Press, 1996.

[Mylopoulos 84] Mylopoulos, John and Levesque, Hector J. "An Overview of Knowledge Representation," 3-17. *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages* (Michael L. Brodie, John Mylopoulos, Joachim W. Schmidt, editors). New York, NY: Springer-Verlag, 1984.

[Nanard 91] Nanard, Jocelyne and Nanard, Marc. "Using Structured Types to Incorporate Knowledge in Hypertext," 329-343. *Proceedings of Hypertext '91.* San Antonio, Texas; December 15-18, 1991. New York, NY: ACM Press, 1991.

[Nielsen 90a] Nielsen, Jacob. *Hypertext & Hypermedia.* Academic Press, 1990.

[Nielsen 90b] Nielson, Jacob. "The Art of Navigating Through Hypertext." *Communications of the ACM 33*, 3 (March 1990): 296-310.

[Ning 89] Ning, Jim Qun. *A Knowledge-Based Approach to Automatic Program Analysis* (Ph.D. thesis). Urbana-Champaign, IL: Department of Computer Science, University of Illinois at Urbana-Champaign, 1989.

[Olsem 93] Olsem, Mike R. and Sittenauer, C. *Reengineering Technology Report, Volume I* (Technical Report). Salt Lake City, UT: Software Technology Support Center, 1993.

[Müller 92] Müller , Hausi A.; Tilley, Scott R.; Orgun, Mehmet A.; Corrie, Brian D.; and Madhavji, Nazim H. "A Reverse Engineering Environment Based on Spatial and Visual Software Interconnection Models," 88-98. *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments (SIGSOFT '92).* Tyson's Corner, Virginia, December 9-11, 1992. *ACM Software Engineering Notes 17*, 5 (1992).

[Parnas 79] Parnas, David L. "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering SE-5*, 2 (March 1979): 128-137.

[Paul 95] Paul, Santanu. *Design and Implementation of Query Languages for Program Databases* (Ph.D. thesis). Ann Arbor, MI: Department of Computer Science and Engineering, University of Michigan, 1995.

[Rohrich 87] Rohrich, J. "Graph Attribution with Multiple Attribute Grammars," *ACM SIGPLAN* 22, 11 (November 1987): 55-70.

[Shaw 89] Shaw, Mary. "Larger Scale Systems Require Higher-Level Abstractions." *ACM SIGSOFT Software Engineering Notes 14*, 3 (May 1989): 143-146.

[Smith 97] Smith, Dennis B.; Müller, Hausi A.; and Tilley, Scott R. *The Year 2000 Problem: Issues and Implications* (CMU/SEI-97-TR-002, ADA 325361). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.

[Sneed 84] Sneed, Harry. "Software Renewal: A Case Study," *IEEE Software 1*, 3 (July 1984): 56-63.

[Sowa 88] Sowa, John F. *Conceptual Structures: Information Processing in Mind and Machine*. Reading, MA: Addison-Wesley, 1988.

[Storey 96] Storey, Margaret-Anne D.; Wong, Kenny; Fong, P.; Hooper, D.; Hopkins, K.; and Müller, Hausi A. "On Designing an Experiment to Evaluate a Reverse Engineering Tool," 31-40. *Proceedings of the 3rd Working Conference on Reverse Engineering*. Monterey, CA, November 8-10, 1996. Los Alamitos, CA: IEEE Computer Society Press, 1996.

[Storey 97a] Storey, Margaret-Anne D.; Fracchia, David; and Müller, Hausi A. "Cognitive Design Elements to Support the Construction of a Mental Model During Software Visualization," 17-28. *Proceedings of the 5th Workshop on Program Comprehension*. Dearborn, Michigan: May 28-30, 1997. Los Alamitos, CA: IEEE Computer Society Press, 1997.

[Storey 97b] Storey, Margaret-Anne D.; Wong, Kenny; and Müller, Hausi A. "How Do Program Understanding Tools Affect How Programmers Understand Programs?" 12-21. *Proceedings of the 4th Working Conference on Reverse Engineering*. Amsterdam, The Netherlands, October 6-8, 1997. Los Alamitos, CA: IEEE Computer Society Press, 1997.

[Tilley 91] Tilley, Scott R. and Müller, Hausi A. "INFO: A Simple Document Annotation Facility," 30-36. *Proceedings of the 9th Annual International Conference on Systems Documentation (SIGDOC '91)*. Chicago, Illinois, October 10-12, 1991. New York, NY: ACM, 1991.

[Tilley 92] Tilley, Scott R.; Müller, Hausi A.; and Orgun, Mehmet A. "Documenting Software Systems with Views," 211-219. *Proceedings of the 10th International Conference on Systems Documentation (SIGDOC '92)*. Ottawa, Canada, October 13-16, 1992. New York, NY: ACM, 1992.

[Tilley 95a] Tilley, Scott R. *Domain-Retargetable Reverse Engineering* (Ph.D. thesis). Victoria, Canada: Department of Computer Science, University of Victoria, January 1995.

[Tilley 95b] Tilley, Scott R. "Domain-Retargetable Reverse Engineering III: Layered Modeling," 52-61. *Proceedings of the 1995 International Conference on Software Maintenance (ICSM '95).* Nice, France, October 17-20, 1995. Los Alamitos, CA: IEEE Computer Society Press, 1995.

[Tilley 96a] Tilley, Scott R.; Santanu, Paul; and Smith, Dennis B. "Toward a Framework for Program Understanding," 19-28. *Proceedings of the 4th Workshop on Program Comprehension.* Berlin, Germany, March 29-31, 1996. Los Alamitos, CA: IEEE Computer Society Press, 1996.

[Tilley 96b] Tilley, Scott R. and Smith, Dennis B. *Coming Attractions in Program Understanding* (CMU/SEI-96-TR-019 ADA320731). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996.

[Tilley 97] Tilley, Scott R. *Discovering DISCOVER* (CMU/SEI-97-TR-012 ADA331014). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.

[Tilley 98] Tilley, Scott R. *Coming Attractions in Program Understanding II: Highlights of 1997 and Opportunities for 1998* (CMU/SEI-98-TR-001). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1998.

[Tracz 94] Tracz, Will. "Domain-Specific Software Architecture (DSSA) Frequently Asked Questions (FAQ)." *ACM SIGSOFT Software Engineering Notes 19*, 2 (April 1994): 52-56.

[von Mayrhauser 92] von Mayrhauser, Anniliese and Vans, Marie. *An Industrial Experience With an Integrated Code Comprehension Model* (Technical Report CS-92-205). Ft. Collins, CO: Colorado State University, 1992.

[von Mayrhauser 95] von Mayrhauser, Anniliese and Vans, Marie. "Program Comprehension During Software Maintenance and Evolution," *Computer 28*, 8. (August 1995): 44-55.

[Ullman 80] Ullman, Jeffrey D. *Principles of Database Systems*. New York, NY: Computer Science Press, 1980.

[Wasserman 89] Wasserman, Anthony I. "Tool Integration in Software Engineering Environments," 137-149. *Proceedings of the International Workshop on Environments*. Chinon, France, September 18-20, 1989. New York, NY: Springer-Verlag, 1989.

[Wong 95] Wong, Kenny; Tilley, Scott R.; Müller, Hausi A.; and Storey, Margaret-Anne D. "Structural Redocumentation: A Case Study," *IEEE Software 12*, 1 (January 1995): 46-54.

[Young 88] Young, Michael; Taylor, Richard N.; and Troup, Dennis B. "Software Environment Architectures and User Interface Facilities." *IEEE Transactions on Software Engineering 14*, 6 (June 1988): 697-708.

[Zvegintzov 94] Zvegintzov, Nicholas (editor). *Software Management Technology—1994 Edition.* Staten Island, NY: Software Management News Inc., 1994.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (LEAVE BLANK)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	April 1998	Final

4. TITLE AND SUBTITLE
A Reverse-Engineering Environment Framework

5. FUNDING NUMBERS
C — F19628-95-C-0003

6. AUTHOR(S)
Scott Tilley

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

8. PERFORMING ORGANIZATION REPORT NUMBER
CMU/SEI-98-TR-005

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
ESC-TR-98-005

11. SUPPLEMENTARY NOTES

12.A DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified/Unlimited, DTIC, NTIS

12.B DISTRIBUTION CODE

13. ABSTRACT (MAXIMUM 200 WORDS)

This report describes a framework for reverse-engineering environments used to aid program understanding. The framework is based on a descriptive model that categorizes important support mechanism features based on a hierarchy of attributes. The attributes include cognitive model support, reverse-engineering tasks, canonical activities that are characteristic of the reverse-engineering process, quality attributes supported by the reverse-engineering environment, and miscellaneous characteristics.

14. SUBJECT TERMS
canonical activities, cognitive aids, conceptual framework, data gathering, descriptive model, environment, information exploration, knowledge management, program understanding, quality attributes, reverse engineering

15. NUMBER OF PAGES
44

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102