The Role of Understanding and Adaptation in Software Reuse Scenarios

Karen E. Huff, Ronnie Thomson, and James W. Gish
GTE Laboratories Incorporated
40 Sylvan Road
Waltham, MA 02254

Internet: jgish@gte.com

Abstract

Many discussions of reuse presuppose a context of new development involving reusable components retrieved from a library. We present an expanded view of reuse that encompasses a spectrum of software evolution scenarios, including original development and maintenance. Central to all these reuse scenarios are the activities of software understanding and adaptation. We discuss the role of domain knowledge in understanding and adaptation, and describe an approach to their support using an explicit representation of this knowledge.

Keywords: software reuse, software adaptation, software understanding, software evolution, reengineering, reverse engineering

1 Introduction

Software reuse has the potential to increase productivity and reduce time to market through savings in design, code and test effort, as well as to improve quality and reduce risk through the use of proven components. A common approach to supporting reuse is to organize software components into a library. The investment in reuse includes acquiring, qualifying, and cataloging suitable components in this library. The payoff is that developers can search the library to retrieve components for new systems.

A library-based process is not the only scenario involving reuse. When maintaining an existing system, that system's current components are the default foundation for change; this view of maintenance as reuse is described in [Basi 90]. Similarly, during incremental development, certain of the present system components are modified and the others reused "as is" to create the next system version. Even within library-based reuse, two overhead processes involve reuse: populating a new reuse library or improving an existing one. Components harvested from existing systems or existing libraries may be reused in the creation of improved components with maximized reuse potential.

Figure 1 shows commonalities and differences among four reuse scenarios: system evolution, library-based reuse, library population, and library improvement. By varying the sources and destinations of components, these cases are revealed:

- *System evolution* - Path 1 to 3: either maintenance or incremental development), assuming source and destination systems (S and T) are the same.

  In the case where source and destination systems are different, this is ad hoc reuse from one system to another.

- *Library population* - Path 1 to 4: harvesting a component from an existing system to put in the reuse library

- *Library improvement* - Path 2 to 4: generalizing or specializing a component already in the library

- *Library-based reuse* - Path 2 to 3: retrieval from the library for reuse in some system.

2 Understanding and Adaptation

In each of these scenarios, understanding and adaptation are central to the process, as the figure suggests. Adaptation is needed when certain design and implementation decisions that were appropriate to the original application prove unsuitable for new applications. It involves factoring out

the inappropriate choices and replacing them; this may in turn affect other choices, and therefore cause additional adaptations. Understanding the structure and interrelationships between various entities in a software component is essential to its adaptation and integration in a new application. It is also important in deciding whether a candidate component is actually suitable in a particular new application.

Software developers are justifiably wary of the effort involved in, and the difficulties inherent to, understanding and adapting software. Since these activities are so central to reuse in all its scenarios, we believe that there is a need for tools that support them directly.

Currently, understanding and adaptation are most frequently carried out using a text editor, whose operations manipulate character strings. The problem is that character strings are a lowest common denominator representation that does not capitalize on the special semantics of the type of text being manipulated. We are developing an alternative approach that explicitly employs knowledge of the application domain, programming constructs, and the mappings between them. Thus, instead of issuing a command to search for a character string, the developer can ask to see the "message encryption functions". And, instead of issuing commands to replace certain character strings with others, the developer can issue commands to "add message acknowledgment" or "substitute a connectionless socket for the connection-oriented socket". These two approaches are contrasted in Figure 2. In addition to differences in the level and power of the commands the developer uses, there are differences in the guarantees that can be made about the adapted component; a component adapted via editing may not even compile, while a component that is

adapted in the alternate way should meet even higher standards than compilability. Our technical approach to achieving this model is described below.

3 The Role of Domain Knowledge

The kind of support for understanding and adaptation that we envision depends on domain knowledge [1] - concepts like message acknowledgement, connectionless sockets, and message encryption for a message processing domain. There is empirical evidence that developers learn, remember, and employ chunks of domain knowledge both in constructing new programs and in understanding existing ones [Solo 84]. There is further evidence that expert developers work with chunks that embody domain-specific knowledge, and that it is this knowledge of mappings between domain concepts and programming concepts that is essential to their expert performance [Broo 87, Curt 87].

The key idea is the explicit representation of this domain knowledge as the deeper structure or pattern map behind a program. This pattern map shows in detail how individual parts, whether disparate statements, contiguous statements, procedures and the like, are interrelated. One way to view a pattern map is as a set of instances of pre-defined patterns of programming/domain knowledge that have been composed according to certain rules about data flow, control flow, and other constraints. A number of recent research projects have attempted to account for the structure of software in just this way, and we intend to build upon this work. These projects include the Programmer's Apprentice [Rich 88], PROUST [John 85], and PAT [Hara 90].

To give an example, there is a pattern that describes one way to reassemble messages from individual packets when packets are expected to arrive out of order. This pattern shows how two abstract data types, one for a translation table and one for random access memory (RAM), are used with specific information from the message itself. At the center of this pattern is the strategy for handling temporary storage of incomplete messages. The message id is used in a call to the translation table "lookup" operation, giving a starting address for temporary storage of the message until all its packets are received. The message sequence number is multiplied by the packet size to give an offset , which is added to the starting address. This new address, along with the content of the current packet, is used in a call to the "store" operation of the RAM data type This pattern, which shows how certain data flows from the message to calls on operations of the two abstract data types, makes a mapping between two computer science concepts (translation table and RAM) and concepts from the message processing domain.

4 Implementation Approach

Our approach to supporting understanding and adaptation is predicated upon a library of predefined patterns, which can be stored in the reuse library along with the reusable software components. Domain analysis techniques can be used to acquire the knowledge embedded in the patterns. Given

[1] In fact, the use of domain knowledge distinguishes what we mean by understanding from reverse engineering, and what we mean by adaptation from reengineering. As currently conceived, both reverse engineering and reengineering tools incorporate the type of knowledge about components that is found in the typical compiler, such as knowledge about name scopes and connections between declarations and uses. A reverse engineering tool that is an exception to this is DESIRE [Bigg 89].

this representation of domain knowledge, understanding can be implemented as the recognition of these patterns, while adaptation can be implemented as the substitution of alternative patterns for certain existing ones. For example, in the message assembly pattern described above, it might happen that, due to new capacity considerations, the space to store incomplete messages should not be assumed to be memory resident. In response, the very simple substitution of a paged memory abstract data type for the RAM can be made.

The architecture for a system to support understanding and adaptation thus has four key components: a representation for patterns capturing domain knowledge, an understanding algorithm for recognizing these patterns in software components, an adaptation algorithm for substituting one pattern for another, and a domain analysis component. This is shown in Figure 3.

This work is motivated by two techniques from AI: planning and case-based reasoning. Planning is concerned with the representation of a rationale for actions in a data structure called a plan. There are similarities between the patterns describing the deep structure of a program and the plan describing the rationale for actions. Case-based reasoning is a framework for using existing solutions to past problems as a way to solve new problems that happen to be similar. There is a direct correspondence with the library-based reuse processQone merely substitutes "case memory" for "library" and "solution" for "software component". Our work on adaptation addresses the problem of modifying existing cases to fit new problems. Modifying the "plan" of an old case serves as the theory for modifying the case itself.

The research described here, which is in an early stage, is directed at processes that are central to all reuse scenarios. It extends the focus of reuse beyond issues of library organization and retrieval. The difficulties of software reuse in the current software development paradigm raise fundamental questions about the nature of software parts and their interconnections. We hope to gain some insight on these questions from developing support for the processes of understanding and adaptation.

References

[Basi 90] Basili, V. "Viewing Maintenance as Reuse-Oriented Software Development." *IEEE Software*, 7:1 (January 1990), 19-25.

[Bigg 89] Biggerstaff, T. "Design Recovery for Maintenance and Reuse." *IEEE Computer*, 22:7 (July 1989), 36-49.

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

[Curt 87] Curtis, W.; Krasner, H.; Shen, V.; and Iscoe, N. "On Building Software Process Models Under the Lamppost." *9th Int'l. Conf. Software Eng.*, Washington, D.C.: IEEE Press, 96-103.

[Hara 90] Harandi, M., and Ning, J. "Knowledge-based Program Analysis." *IEEE Software*, 7:1 (January 1990), 74-81.

[John 85] Johnson, W.L. and Soloway, E. "PROUST: Knowledge-based Program Understanding." *IEEE Trans. Software Eng.*,11:3 (March 1985), 267-275.

[Rich 88] Rich, C. and Waters, R. "Programmer's Apprentice: A Research Overview." *IEEE Computer* , 21:11 (November 1988), 10-25.

[Solo 84] Soloway, E. and Ehrlich, K. "Empirical Studies of Programming Knowledge." *IEEE Trans. Software Eng.*, 10:5 (September 1984), 595-609.

5 About the Authors

Jim Gish is a Senior Member of Technical Staff at GTE Laboratories and a member of the Software Reusability Project and served as Principal Investigator of that project from 1989-1990. Before joining the Software Technologies Department at GTE in 1987, he was a Principal Software Engineer at Prime Computer where he lead the design of a CCITT X.400 Computer Based Message System. From 1982 to 1985, Mr. Gish was a Senior Systems Programmer and Software Development Manager at Datapoint Corporation where he contributed to the design and development of Datapoint's Vistamail Electronic Mail System. Mr. Gish holds a BS degree in Statistics and Computer Science from the University of Delaware, and an MS degree in Computer Sciences from the University of Wisconsin-Madison. He is ABD from the University of Wisconsin-Madison.

Karen E. Huff has over 20 years of experience in software research and development. Her research interest is the application of AI techniques to problems in software engineering. In her dissertation, she developed an architecture for intelligent assistance of software development processes, by integrating AI planning and plan recognition techniques. From 1977-1986, she was with Intermetrics, Inc. of Cambridge, MA. In addition to serving as a department head, she managed projects involving optimizing compilers, other software tools, and integrated software environments. Prior to joining Intermetrics, Dr. Huff had eight years of software development experience, including custom

software development at a university computing center and both compiler and operating system projects at Control Data Corporation. Her Ph.D. is from University of Massachusetts at Amherst, M.S. from Stanford University, and B.A. from Oberlin College. She joined GTE Laboratories in 1989.

Ronnie Thomson is a member of the Software Reusability Project at GTE Laboratories. His principal interests lie in the area of a reuse-centered software development process and in the identification of tools and methods which support the process of reuse. From 1984-1987 he was with the University of Strathclyde, Glasgow, Scotland, working on the development of a module interconnection language and toolset supporting configuration management activities in the Eclipse software engineering environment. From 1987-1989 he worked on the European Esprit project, Dragon, at the University of Lancaster, England, which developed tools and methods for designing reusable software for distributed real-time applications. His B.Sc. and Ph.D. are from the University of Strathclyde. He joined GTE Laboratories in 1989.