Reuse in Software Engineering: An Object-Oriented Perspective

Kenneth S. Rubin

ParcPlace Systems, Inc.
1550 Plymouth Street
Mountain View, CA 94043

Stanford University
Department of Computer Science
Stanford, CA 94305

Abstract

This paper examines the role of reuse within software engineering. The benefits of reuse are stated, inhibitors to succesful reuse are pointed-out, and technical issues for achieving reuse are examined. Specific attention is paid to how object-oriented software engineering enhances or reduces the potential for reuse as compared with traditional* software engineering.

Introduction

It is widely believed that the development and utilization of reusable software artifacts is necessary for improving software development productivity and software quality [1, 2, 3, 4, 5, 6, 7]. Most software development methodologies recognize the utility of reuse, and some even provide processes and tools to directly support it [1, 2, 4]. An emerging object-oriented software engineering methodology is similarly grounded in the utility and benefits of the development and reuse of software artifacts. The purpose of this paper is to explore software engineering reuse, and in particular, the relationship between reuse and object-oriented software engineering.

Software Engineering

Software engineering is concerned with the notations, processes, tools, and artifacts used by a team of individuals working jointly on the development of a large software endeavor. Answers to a recent survey on software engineering conducted by ParcPlace Systems indicate that what constitutes a large project differs between different individuals or organizations. For the purpose of this paper, a large project is assummed to be one that is staffed by at least two people, and whose complexity is such that a single individual is not capable of fully comprehending every aspect of the system under development.

In order to describe the requirements of a software engineering methodology, it is instructive to examine the characteristics of large systems to be designed using the methodology. Large systems are built by many people working together. A software engineering methodology should provide processes and tools for dealing with the complications of multiple people simultaneously working on a set of related tasks with shared information.

The development costs of large systems are often high due to both the size of the project and the necessity multi-person teams. These costs could be reduced if some of the components used to build the system are taken or reused from existing systems. For this reason, a software engineering methodology should directly support the notion of developing and leveraging off of reusable software artifacts.

Large, complex, high-cost systems that employ many people are usually long-lived. Over their lifetime, such systems will almost certainly evolve to accommodate changes in the original specification. An effective methodology should acknowledge the fact that system evolution is inevitable, and that adaptation to change is a fundamental necessity.

* In the context of this paper, the term "traditional" is used to qualify those notations, processes, tools, and artifacts that are not based on objects or object-oriented concepts.

Since large systems are often too complicated for any one individual to comprehend, it becomes extremely difficult for users to specify exactly what is wanted. In such circumstances, the system specification is often incomplete and inaccurate. A scaled-down prototype can serve as a vehicle by which users and systems analysts discover true user requirements. A prototype is also useful during the design phase as a means of achieving architectural clarity. A software engineering methodology should support the notion of prototyping.

The software engineering field currently utilizes several well-defined traditional methodologies, e.g., Structured Analysis, Structured Design, and Jackson Design Method, etc. To some extent, each of these supports the previously specified requirements. However, it is not clear to what extent, if any, these techniques are useful for the development of object-oriented systems. With the increasing popularity of object-oriented technology, it is important to consider what is necessary to support the software engineering requirements described above.

To date, there is no complete object-oriented software engineering methodology that specifies a set of well-defined notations, processes, tools, and artifacts that are used by teams of individuals working jointly on the development of large object-oriented software systems. However, developing such a methodology is an active focus of contemporary research [8,9].

Definition of Object-Oriented Software Engineering

There have been many attempts to define object-oriented systems. For our purpose, a very modest definition of object-oriented software engineering suffices: object-oriented software engineering is the process of system development that encompasses notations, processes, tools, and artifacts, based on the premise that a domain of discourse may be modeled by a collection of interacting objects. An object is defined as an encapsulation of state and behavior within a single software entity that reflects a corresponding entity in some real domain. The state of an object represents the internal information content of the object (i.e., its properties or attributes). This information is not directly accessible from outside of the object. The behavior of an object is the set of activities or operations that an object can perform, including answering questions about its state.

The metaphor for work in object-oriented systems is one of objects communicating with one another via messages. Messages are the medium by which one object may instruct another object what to do, but not how to do it. Upon receiving a message, an object invokes one of its behaviors (this can be viewed as a function or procedure call in traditional language). The set of all messages to which an object can respond is known as the object's protocol, and this protocol represents the external interface through which other objects may evoke its behavior.

In order to derive the object abstractions for a system, object-oriented analysis and design are used. No complete and validated methodology for object-oriented analysis and design has yet been described, however, there are some very useful techniques that have been applied to the development of large systems.

All these techniques are based on the fundamental properties of object-oriented systems: encapsulation, polymorphism, and inheritance. While none of these properties is unique to object-oriented systems (many traditional systems support encapsulation, some support polymorphism, fewer support inheritance), only object-oriented systems have first-class support for all of them. We now briefly describe each of these properties within the context of object-oriented systems.

Encapsulation. Encapsulation is the combining of state and behavior into objects such that clients of an object can specify what operations an object should perform, but have no knowledge of how the object carries out the operations. Encapsulation is analogous to the abstract data types supported by Modula-2 modules and Ada packages, and carries the same positive connotations of information and implementation hiding.

Inheritance. Inheritance is the mechanism by which objects may share definitions of state and behavior. To do so, common state and behavior may be placed in one object and inherited by other objects. This has the benefit of eliminating duplicate information, which reduces the effort necessary for debugging, testing, and maintenance, and provides for conceptual clarity. In addition, a special style of design, known as design by refinement (specialization), may be supported. Although some traditional systems simulate inheritance (most notably some of the LISPs), none of them support inheritance as a first class property.

Polymorphism. Polymorphism is the ability to reuse the same names to evoke potentially different behavior. This is referred to as overloading in languages such as Ada. The benefits of using polymorphism include reduction in the overall number of names required, plus the ability to design substitution-compatible objects. Such objects support the same external interface (protocol and generic behavior), but have potentially different internal implementations. The utility of substitution-compatible objects is that they may be interchanged with one another (if semantically appropriate) because they support the same protocol.

What is Reuse?

The essence of reuse is the notion of using what already exists in order to achieve what is desired. In the context of software engineering, there are many artifacts that may potentially be reused, some of which include analyses, designs, implementations, test cases, and documentation. Henceforth the word artifact is used to refer to any of these potentially reusable entities, and context will disambiguate which particular artifact, if any, is being discussed.

The motivation for reuse is based primarily on productivity and quality improvements. Productivity can be viewed as a function of cost and effort. If reusing an artifact is of less cost and effort than developing it from scratch, productivity is improved. The extent to which reusable artifacts have been tested and proven reliable, will directly and proportionally improve the quality of the system that is reusing them. One might summarize the goal of reuse as development of a set of abstractions (artifacts) which are not limited to a single software effort, and can be used to make future software systems development more productive while simultaneously enhancing productivity.

The benefit of reusing an artifact is related to the artifact's abstraction level. The higher the level of abstraction the higher the potential payoff [6]. Abstract artifacts have limited detail and little reliance on implementation decisions. This enables their reuse during early stages of a project when it is most likely to have the largest widespread impact. Analyses (i.e., specifications) and designs are the most abstract reusable artifacts, and have the highest potential payoff in terms of productivity and quality. Implementation reuse, clearly the most commonly practiced form of reuse to date [1, 2, 3, 5], does not have as great a potential payoff as specification and design reuse. Biggerstaff, and others, point out that the amount of a system that can be formed from reusable implementation artifacts is usually under half the system [6].

There are several reasons, both technical and social, why achieving software system reuse is difficult and not commonly practiced. The first reason is the lack of a well-defined formal representation for reusable artifacts. Another is the immaturity of tools and techniques for supporting software reuse. Still another is the lack of an economic model which explains to software development organizations what the real benefits and costs of software reuse are, and which provides a method to analyze their situation [4]. Finally, there is the lack of a corporate infrastructure which encourages and rewards reuse [4].

To effectuate reuse, three major engineering activities must be addressed. First, reusable artifacts must be intentionally designed and developed. Second, reusable artifacts must be represented, classified, and entered into (and removed from) appropriate repositories (i.e., libraries). Third, tools and processes must be developed that support finding, understanding, modifying, and composing artifacts. Each of these is now explored.

Development of Reusable Artifacts

At start of using any new technology there are no reusable artifacts. Individuals and organizations must make a commitment to develop them. Reuse does not happen by accident, rather it requires planned engineering activities. These activities require extra intellectual power, extra money, extra time, and managerial approval to support all the extras.

Designing reusable artifacts is technically challenging, with many factors and trade-offs to consider. For example, choosing the proper abstraction level of an artifact is important. The more specific the artifact, the less likely it is to be generally reusable. The problem is the degree to which the visible attributes of an artifact reflect a specific application domain. In general, the breadth of applicability of an artifact diminishes with domain specificity. On the other hand, generalized artifacts (i.e., ones that make little or no assumptions on domain knowledge), tend to be more reusable, but usually at the expense of time or space or both.

Complexity of an artifact is another dimension. Complex artifacts have many aspects, each of which must be matched in part or in whole in order for reuse to be

possible. Hence, artifacts with many aspects are often more difficult to reuse than artifacts with fewer aspects. Choosing the appropriate complexity of an artifact is often non-trivial. To some extent it requires 20/20 hindsight of past project requirements and 20/20 foresight as to future project requirements. This is clearly an area where experience is very important.

Object-Oriented Design

In an object-oriented system, the fundamental unit of implementation reuse is the object (i.e., classes in class-based object-oriented systems [1], and prototypes in prototype-based object-oriented systems [10]). Related objects may be grouped together to form frameworks and toolkits. Development and use of these reusable artifacts is supported by the properties of object-oriented systems described earlier, and four general-purpose design techniques: refinement, composition, abstraction, and factorization.

Refinement. Design by refinement (or specialization) is a technique for developing new objects from existing objects. This technique, sometimes referred to as programming-by-difference, defines a new object by specifying how it semantically differs from an existing object. To do so, a new object inherits state and behavior from an existing object, and then either refines or adds state and behavior to describe how it is different.

Composition. Design by composition is a technique that creates a new object by combining several existing objects. The new object reuses both state and behavior of the original objects by sending messages to them in order to invoke behavior and access state information. This process is often referred to as delegation.

Both refinement and composition make extensive use of preexisting object definitions. However, because refinement yields more specialized components, and composition yields more complex components, these two techniques do not directly support development of new reusable object definitions. There are, however, two design techniques, abstraction and factorization, which take specialized objects and redefine them to be more general, and hence more reusable.

Abstraction. Design by abstraction is a two-step technique. In the first step, a new object is created that contains the common state and behavior of a collection of specific objects. In the second step, the original objects are redefined (using design by refinement) to inherit from the newly created object. The result is a hierarchy of objects with the new object (as parent) containing the common state and behavior, and the original objects (as children) inheriting this state and behavior.

By using design by abstraction, the common state and behavior that was replicated in the original objects now exists in only one place (the new object). This improves localization and reduces duplication which leads to better maintenance and extensibility. In addition, there was an implicit abstraction hidden in the original objects. This abstraction was made explicit by creating a new object that embodies it. Future designers can use the new object containing the explicit abstraction, rather than having to discover the abstraction for themselves.

Factorization. Design by factorization takes a complex specific object and creates a collection of less complex more general objects that are potentially more reusable. Like abstraction, factorization is also a two-step technique. In the first step, a large, complex object is broken up into several smaller, less complex objects (based on such factors as cohesions of properties and behavior). These new objects are integrated into the hierarchy of existing objects. The second step requires that a new object be designed that has the capabilities of the original object. This new object is created by using the smaller factored objects and the techniques of refinement or composition or some combination of the two. For excellent examples of design by factorization, abstraction, composition, and refinement, the interested reader is referenced to [11].

Representing Reusable Artifacts

The most difficult problem with reuse is developing a suitable representation for artifacts. In particular, we would like a representation that encodes the semantics of artifacts in such a way that a user who is trying to solve a problem with his own knowledge and semantics can locate an appropriate reusable artifact. Such artifacts should be retrievable by multiple pathways to support the variety of different ways in which users may access them. Furthermore, the representation should allow for a variety of different perspectives on stored artifacts, and should permit versioning and configuration management activities. Lastly, a representation should allow for partial and uncertain information. This allows artifact developers to evolve their designs over time by permitting well-defined aspects to be expressed with certainty, and less well-defined

aspects to be left fuzzy.

Once a representation scheme is chosen, a storage and retrieval system must be chosen. Most users prefer to use declarative queries whereby they specify what they want, not how to go about finding it. There are many systems that support declarative queries, but these are not well-adapted at responding to imprecise queries or searching over partial or uncertain information. What is needed is a retrieval system that contains a minimal reasoning mechanism to handle these situations. In general, we would like users to specify what they want to the best of their ability, and have the system retrieve an artifact that is either an exact solution, a partial solution that can be refined, or a set of solutions that can be combined.

Once artifacts are represented, they must be classified and entered into repositories. Classification of artifacts is an indexing issue. As such, artifacts are classified in order to indicate their type and relation to other artifacts. There are two well known schemes for doing this repository classification: enumerative and faceted [7]. The enumerative scheme divides the universe up into a collection of domains and subdomains. This is the approach taken by the Dewey Decimal system used by U.S. libraries. A new artifact is classified by choosing the predefined domain that best describes it. The faceted approach does not rely on an *a priori* division of the universe into domains, but rather synthesizes a classification of an artifact based on the selection of properties from a collection of facets. The facets are the dimensions along which it is useful to classify artifacts.

In practice, the problem of choosing a proper representation and classification scheme has proven quite difficult. For example, Biggerstaff points out that the largest inhibitor to successful design reuse is the difficulty in developing a formal design representation scheme [6]. Burton [2], Arnold [5], Frakes [3], and Prieto-Diaz [7] have all studied the problem of implementation representation, and have used either one, or a combination of both of the classification schemes yielding noticeably different conclusions.

Object-oriented software engineering has potential for making the representation problem tractable. This is due in part to object encapsulation and polymorphic protocols. Object encapsulation simplifies the semantics of reasoning about what an object does by reducing the scope of the problem to that of reasoning about the object's external interface (i.e., protocol) without having to reason about its implementation. Polymorphic protocols provide a criteria upon which a classification scheme can index similar objects. Objects with similar protocols may be substitution-compatible if other semantic criteria are met. Hence, the classification scheme can use protocols to help index potentially similar objects in a manner such that the retrieval system can identify similar and analogous solutions.

Although encapsulation and polymorphism may simplify the representation problem, inheritance may add complications. Recall that inheritance is useful for sharing state and behavior. This sharing leads to distributed object definitions. The following example illustrates the situation. Object B inherits from object A. Since B inherits from A, it is no longer sufficient to examine just the definition of object B in order to completely understand B. Rather, to completely understand object B it is necessary to examine object A to determine inherited state and behavior. This same argument applies to A. It is necessary to examine A's parent object to determine the state and behavior that A inherits. The chain ends when the root of the inheritance hierarchy is reached.

The problem created by inheritance is how to represent the implementation of an object when it is necessary to describe the implementation of its parent objects. There are several solutions to this problem; each one will certainly complicate the representation.

Supporting the Reuse of Artifacts

Once artifacts have been developed, represented, and categorized into repositories, the next interesting issue is how to utilize this wealth of information. Software developers need tools and processes for finding, understanding, and using reusable artifacts.

Finding Artifacts

To find artifacts users describe what they want and tools decide how to satisfy the request. This simple declarative model is rarely achieved in practice. The simple reason is that most representations are insufficient to support sophisticated queries and reasoning. For example, the UNIX grep command is one way of finding artifacts is a file-based scheme whose representation is textual. However, it is up to the user to interpret the file and line information returned as a result of the query.

The Smalltalk-80 System Browser [1] is a sophisticated tool for locating artifacts (i.e., classes and meth-

ods) based on a four level hierarchy of structured navigation. However, this tool requires that the user drive the navigation process. Not only must the user know what he wants, he must also know how to go about finding it.

Another sophisticated example is the Software Component Retrieval and Evaluation (SCORE) subsystem of the RSL system [2]. The RSL couples a passive database with interactive design tools that use a library of reusable code artifacts that are categorized by 14 different attributes. SCORE selects and evaluates artifacts based on the designer's responses to questions about his software requirements. The output of SCORE is an ordered list of recommended artifacts.

The sophistication of techniques for finding information is dictated by the representation scheme. Hence, the extent to which object-oriented software engineering is more or less able to support retrieval than traditional software engineering will be dictated by the representation scheme. To date, a sophisticated representation for object-oriented software engineering artifacts has yet to be devised. The Smalltalk-80 system, considered by many to include the best library navigation tool available (i.e., the System Browser), could be significantly improved if the underlying representation and classification of the objects were enhanced to support a broader and deeper range of information about the objects.

Understanding Artifacts

Once artifacts have been located, it is necessary to understand them in order to use them. People cannot be expected to use or modify something they cannot understand. This a fundamental problem. In fact, one of the biggest inhibitors to successful reuse is the inability of individuals to understand (and trust) the work of one another. To overcome this, tools are needed that permit the user to probe the retrieved artifact by inspecting it in a variety of ways, asking questioning about it, cross referencing its components, and finding examples of its use. A technology that may hold promise in this realm is hypermedia. The ability to create webs of interconnections among various textual and graphical documents enable individuals to directly link together a plethora of information. However, a drawback to this approach is that it does not directly support declarative queries.

Object-oriented software engineering has potential to enhance the understandability of software artifacts. A strength of an object-oriented approach to development is that it offers a mechanism (objects) that captures a model of the real world. Previous research by cognitive psychologists in the area of classification theory has shown that the notion of concepts and classification adds stability to understanding the world because it facilitates a person's ability to understand an entity or situation given very little information [12]. Recent efforts have attempted to show the correlation between classification theory and object-oriented development [13].

There is some empirical evidence that indicates object-oriented technology may enhance software understandability. Specifically, observations made by the author over the past year and half while teaching the ParcPlace Systems Object-Oriented Programming (Smalltalk-80) Courses has indicated that students who learned the fundamentals of object-oriented design and programming, and proper usage of the Smalltalk-80 development tools, were effective in their efforts to find and understand a variety of reusable objects (ranging from simple number and collection-type objects, to sophisticated graphical user interface objects) contained in the Smalltalk-80 library. However, no formal experiments have been performed to support these observations.

Using Artifacts

After retrieving and examining an artifact, a developer may decide to use it. This activity should be viewed as a fundamental part of the development process (e.g., design process, implementation process). As such, development tools and processes must recognize and support the notion of incorporating reusable artifacts taken from a repository.

There are a variety of different ways in which an artifact may be reused. A retrieved artifact that is useful without modification need only be integrated "plugged" into the system. However, if an artifact requires modification, it may be necessary to refine or compose it.

An artifact retrieved from a repository may have most, but not all of the necessary capabilities. In such cases the retrieved artifact may be refined to yield the desired artifact.

In the process of locating suitable artifacts in the repository, the system may not be able to locate a single artifact that meets all of the retrieval requirements. However, it may find a collection of artifacts whose combination is sufficient. In this case, combining the

retrieved artifacts is required.

In an object-oriented system, the refinement and composition tasks are potentially simpler. By using inheritance, refinement is a top-down process of specifying the differences between the inherited state and behavior of an existing object and the requirements of the desired object. With first-class support of encapsulation and message protocols, composition is a bottom-up process of connecting together the proper object building blocks to form the desired component. This is analogous to constructing digital circuits by connecting various integrated circuits together based on their pin specifications.

Conclusions

This paper has focused on reuse within software engineering, with particular emphasis on object-oriented software engineering. Three major activities for supporting reuse were described. The first was the intentional development of reusable artifacts, second was the representation and classification of artifacts into repositories, and last was the utilization of the artifacts from repositories. The object-oriented model was seen to be significantly similar to the traditional model on many aspects, but different on a few. In particular, object-oriented design supports the design techniques of abstraction and factorization that utilize object inheritance to yield reusable objects. Furthermore, encapsulation and polymorphism were seen as potentially useful for addressing the representation and classification problems. On the other hand, object inheritance was seen as a complicating factor when dealing with representation. Nonetheless, object-oriented software engineering may yield models which are conceptually closer to the real domain of discourse, hence making the understanding of reusable components potentially simpler. Lastly, object-oriented inheritance can simplify the refining of artifacts, and encapsulation and polymorphism can simplify the composing of artifacts.

References

[1] A. Goldberg, *Smalltalk-80: Interactive Programming Environment*. Addison-Wesley, Reading, MA, 1983.

[2] B. A. Burton *et al.*, "The Reusable Software Library," Tutorial: Software Reuse: Emerging Technology, IEEE Computer Society, EHO278-2, pages 129-137.

[3] W. B. Frakes and B. A. Nejmeh, "An Information System for Software Reuse," Tutorial: Software Reuse: Emerging Technology, IEEE Computer Society, EHO278-2, pages 142-151.

[4] B. Barnes *et al.*, "A Framework and Economic Foundation for Software Reuse," Tutorial: Software Reuse: Emerging Technology, IEEE Computer Society, EHO278-2, pages 77-88.

[5] S. Arnold and S. Stepoway, "The Reuse System: Cataloging and Retrieval of Reusable Software," Tutorial: Software Reuse: Emerging Technology, IEEE Computer Society, EHO278-2, pages 138-141.

[6] T. Biggerstaff and C. Richter, "Reusability Framework, Assessment, and Directions," Tutorial: Software Reuse: Emerging Technology, IEEE Computer Society, EHO278-2, pages 3-11.

[7] R. Prieto-Diaz and G. A. Jones, "Breathing New Life into Old Software," Tutorial: Software Reuse: Emerging Technology, IEEE Computer Society, EHO278-2, pages 152-160.

[8] G. Booch, "Object-Oriented Development," IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, February 1986, pp. 211-221.

[9] R. Wirfs-Brock and B. Wilkerson, "Object-Oriented Design: A Responsibility-Driven Approach," OOPSLA '89 Conference Proceedings, Special Issue of SIGPLAN Notices, Vol. 24, No. 10, October 1989, pp. 71-76.

[10] D. Ungar and R. B. Smith, "Self: The Power of Simplicity," OOPSLA '87 Conference Proceedings, Special Issue of SIGPLAN Notices, Vol. 22, No. 12, October 1987, pp. 227-242.

[11] ParcPlace Systems, *Objectworks for Smalltalk-80 Tutorial Guide*. ParcPlace Systems, Inc. Mountain View, CA, 1989.

[12] E. E. Smith and D. L. Medin, *Categories and Concepts*. Harvard University Press, Cambridge, MA, 1981.

[13] M. B. Rosson and E. Gold, "Problem-Solution Mapping In Object-Oriented Design," OOPSLA '89 Conference Proceedings, Special Issue of SIGPLAN Notices, Vol. 24, No. 10, October 1989, pp. 7-10.