Reusability of the Software

Sarbjeet Singh, Sukhvinder Singh, Gurpreet Singh

M.Tech. CSE(1st Year)
Sri Sai College of Engg. And Technology, Pathankot, India.

ABSTRACT

Reusability is the likelihood a segment of source code that can be used again to add new functionalities with slight or no modification. Reusable modules and classes reduce implementation time, increase the likelihood that prior testing and use has eliminated bugs and localizes code modifications when a change in implementation is required. Subroutines or functions are the simplest form of reuse. A chunk of code is regularly organized using modules or namespaces into layers. Proponents claim that objects and software components offer a more advanced form of reusability, although it has been tough to objectively measure and define levels or scores of reusability. Reusability implies some explicit management of build, packaging, distribution, installation, configuration, deployment, maintenance and upgrade issues. If these issues are not considered, software may appear to be reusable from design point of view, but will not be reused in practice. This paper presents an empirical study of the software reuse activity by expertdesigners in the context of object-oriented design. Our study focuses on the three following aspects of reuse : (1) the interaction between some design processes, e.g. constructing a problem representation, searching for and evaluating solutions, and reuse processes, i.e. retrieving and using previous solutions, (2) the mental processes involved in reuse, e.g. example-based retrieval or bottom-up versus top-down expanding of the solution, and (3) the mental representations constructed throughout the reuse activity, e.g. dynamic versus static representations.

1. INTRODUCTION

Software permeates our daily life. There is probably no other human-made material which is more omnipresent than software in our modern society. It has become a crucial part of many aspects of society: home appliances, telecommunications, automobiles, airplanes, shopping, auditing, web teaching, personal entertainment, and so on. In particular, science and technology demand high-quality software for making improvements and breakthroughs. Software Reuse is currently one of the most active and creative research areas in Computer Science. First, we analyse how some design processes, e.g. constructing a problem representation, searching for and evaluating the solution(s), and reuse processes, i.e. retrieving and using previous solution(s), may interact. For example, recalling solutions may lead to a revision of the currently-developed solution[1] and retrieving a past solution may produce the addition of constraints to the representation of the current design problem. This combination proves to be effective because it unites a goal refinement and classification strategy with a packing strategy provided by aspect-oriented programming, making use of well-defined relations among functional and quality fragments, we provide mechanisms for weaving those fragments together. We define a coherent process that uses an asset library to find quality characteristics and apply those to a software functional description.

2. TYPES OF REUSE

- **Opportunistic reuse -** While getting ready to begin a project, the team realizes that there are existing components that they can reuse.
- **Planned reuse -** A team strategically designs components so that they'll be reusable in future projects.

Opportunistic reuse can be categorized further:

- Internal reuse - A team reuses its own components. This may be a business decision[2], since the team may want to control a component critical to the project.
- External reuse - A team may choose to license a third-party component. Licensing a third-party component typically costs the team 1 to 20 percent of what it would cost to develop internally.The team must also consider the time it takes to find, learn and integrate THE COMPONENT.

3. RESUE –BASED SOFTWARE ENGINEERING

3.1 Application system reuse

• The whole of an application system may be reused either by incorporating it without change into other systems[3] (COTS reuse) or by developing application families.

3.2 Component reuse

• Components of an application from sub-systems to single objects may be reused.

3.3 Object and function reuse

• Software components that implement a single welldefined object or function may be reused.

4. SOFTWARE RESUE BENEFITS

4.1 Increased dependability

Reused software, that has been tried and tested in working systems, should be m ore dependable than new software. The initial use of the software reveals any design and

implementation faults. These are then fixed, thus reducing the number of failures when the software is reused.

4.2 Reduced process risk

If software exists, there is less uncertainty in the costs of reusing that software than in the costs of development. This is an important factor for project management as it reduces the margin of error in project cost estimation. This is particularly true when relatively large software components such as sub-systems are reused.

4.3 Effective use of specialists

Instead of application specialists doing the same work on different projects[3],[4], these specialists can develop reusable software that encapsulate their knowledge.

4.4 Standards compliance

Some standards, such as user interface standards, can be implemented as a set of standard reusable components. For example, if menus in a user interfaces are implemented using reusable components, all applications present the same menu formats to users. The use of standard user interfaces improves dependability as users are less likely to make mistakes when presented with a familiar interface.

4.5 Accelerated development

Bringing a system to market as early as possible is often more important than overall development costs. Reusing software can speed up system production because both development and validation time should be reduced.

5. RESUE PROBLEMS

5.1 Increased maintenance costs

If the source code of a reused software system or component is not available then maintenance costs may be increased as the reused elements of the system may become increasingly incompatible with system changes.

5.2 Lack of tool support

CASE toolsets may not support development with reuse. It may be difficult or impossible to integrate these tools with a component library system. The software process assumed by these tools may not take reuse into account.

5.3 Not-invented-here syndrome

Some software engineers sometimes prefer to re-write components as they believe that they can improve on the reusable component. This is partly to do with trust and partly to do with the fact that writing original software is seen as more challenging than reusing other people's software.

5.4 Creating and maintaining a component library

Populating a reusable component library and ensuring the software developers can use this library can be expensive. Our current techniques for classifying, cataloguing and retrieving software components are immature.

5.5 Finding, understanding and adapting reusable components

Software components have to be discovered in a library, understood and, sometimes, adapted to work in a n ew environment. Engineers must be reasonably confident of finding a component in the library before they will make routinely include a component search as part of their normal development process.

6. THE REUSE LANDSCAPE

- Although reuse is often simply thought of as the reuse of system components, there are many different approaches to reuse that may be used[5].
- Reuse is possible at a range of levels from simple functions to complete application systems.
- The reuse landscape covers the range of possible reuse techniques.

7. REUSE APPROACHES

Design patterns:- Generic abstractions that occur across applications are represented as design patterns that show abstract and concrete objects and interactions.

Component-based development:-

Systems are developed by integrating components (collections of objects) that conform to component-model standards.

Application frameworks:-Collections of abstract and concrete classes that can be adapted and extended to create application systems.

Legacy system wrapping:-Legacy systems that can be 'wrapped' by defining a set of interfaces and providing access to these legacy systems through these interfaces[6].

Service-oriented systems:-Systems are developed by linking shared services that may be externally provided.

Application product lines:- An application type is generalised around a common architecture so that it can be adapted in different ways for different customers.

COTS integration:- Systems are developed by integrating existing application systems.

Configurable vertical applications:-A generic system is designed so that it can be configured to the needs of specific system customers**.**

Program libraries:-Class and function libraries implementing commonly-used abstractions are available for reuse.

Program generators:- A generator system embeds knowledge of a particular types of application and can generate systems or system fragments in that domain.

Aspect-oriented software development:-Shared components are woven into an application at different places when the program is compiled.

8. DESIGN PATTERN

In software engineering, a design pattern is a general reusable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.Design patterns reside in the domain of modules and interconnections.

At a higher level there are Architectural patterns that are larger in scope, usually describing an overall pattern followed by an entire system. Not all software patterns are design patterns[7].

9. SOFTWARE FRAMEWORK

A software framework, in computer programming, is an abstraction in which common code providing generic functionality can be selectively overridden or specialized by user code providing specific functionality. Frameworks are a special case of software libraries in that they are reusable abstractions of code wrapped in a well-defined API, yet they contain some key distinguishing features that separate them from normal libraries.

Software frameworks have these distinguishing features that separate them from libraries or normal user applications:

1. inversion of control - In a framework, unlike in libraries or normal user applications, the overall program's flow of control is not dictated by the caller, but by the framework.[1]
2. default behavior - A framework has a default behavior. This default behavior must actually be some useful behavior and not a series of no-ops.
3. extensibility - A framework can be extended by the user usually by selective overriding or specialized by user code providing specific functionality
4. non-modifiable framework code - The framework code, in general, is not allowed to be modified. Users can extend the framework, but not modify its code.

10. SYSTEMATIC SOFTWARE REUSE

Systematic software reuse is still the most promising strategy for increasing productivity and improving quality in the software industry. Although it is simple in concept, successful software reuse implementation is difficult in practice. A reason put forward for this is the dependence of software reuse on the context in which it is implemented[8]. Some problematic issues that needs to be addressed related to systematic software reuse are :-

- A clear and well-defined product vision is an essential foundation to an SPL.
- An evolutionary implementation strategy would be a more pragmatic strategy for the company.
- There exist a need for continuous management support and leadership to ensure success.
- An appropriate organizational structure is needed to support SPL engineering.
- The change of mindset from a project-centric company to a product-oriented company is essential.

11. SOFTWARE REUSE IN INDUSTRY

Many organizations have been successful with software reuse. Hewlett-Packard (HP), for example, has along history with different levels of software reuse started in 1989. It began with the development and networked distribution of family instrument modules and evolved in 1991 to a corporate reuse program that guided several divisional reuse pilot projects for embedded instrument and printer firmware[9].

AT&T's BaseWorkX reuse program started in 1990 as an internal, large-grain component-reuse and software bus technology to support telephone-billing systems. By1995, AT&T was reusing 80 to 95% of its components. A more recent example was implemented by ISWRIC (Israel Software Reuse Industrial Consortium), a joint project of seven leading Israeli industrial companies .It was a two-phase, three-year project started in 2000. During the first phase, a common software reuse methodology was developed to enable software developers to systematically evaluate and compare all possible alternative reuse scenarios. During the second phase, all seven participating companies implemented the methodology in real projects. Each company modified the model to better fit the specific needs of its pilot projects and evaluated the methodological aspects relevant to the pilot projects and the company. Another industrial application for software reuse that is becoming more popular is global software distribution system development (GDSD). Skandia, one of the world's top life insurance companies, collaborated with Tata Consultancy Services (TCS) to create several Itenabled financial services in different countries by integrating components developed independently by TCS and other third-party vendors at their own sites . The appearance of some promising software reuse tools may lead to more successful industrial software reuse stories. One example is Code Smith, the software generating tool that can produce code for any text-based language including C#, VB.NET, Java and FORTRAN. The tool has had a huge success in industry due to its advanced integrated development environment and its extensible, template-driven architecture that givedevelopers full control over the generated code.

12. SOFTWARE REUSE AND SEMANTIC WIKIS

Before describing the proposed system components, we introduce some of the important concepts and terminologies that will be used later in the discussion.

A. Wikis

In general, a Wiki is a web application designed to support collaborative authoring by allowing multiple authors to add, remove, and edit content. The word "Wiki" is a shorter form of "Wiki Wiki Web" derived from the Hawaiian expression "Wiki Wiki" which means "quick"[10]. Wiki systems have been very successful in enabling non-technical users to create Web content allowing them to freely share information and evolve the content without rigid workflows, access restrictions, or predefined structures. Throughout the last decade, Wikis have been adopted as collaborative software for a wide verity of uses including software development, bug tracking systems, collaborative writing, project communication and encyclopedia systems2. Regardless of their purpose, Wikis usually share the following characteristics:

Easy Editing: Traditionally, Wikis are edited using asimple browser interface which makes editing simple and allows to modify pages from anywhere with only minimal technical requirements.

Version Control: Every time the content of Wikis is updated, the previous versions are kept which allows rolling back to earlier version when needed.

Searching: Most Wikis support at least title search and some times a full-text search over the content of all pages.

Access: Most Wikis allow unrestricted access to their contents while others apply access restrictions by assigning different levels of permissions to visitors to view, edit, create or delete pages.

Easy Linking: Pages within a Wiki can be linked by their title using hyperlinks.

Description on Demand: Links can be defined to pages that are not created yet, but might be filled with content in the future. From the software reuse point of view, Wikis can be seen as a "lightweight platform for exchanging reusable artifacts between and within software projects" that has a low technical usage barrier. However, using Wiki systems as a knowledge repository for software reuse has a major drawback. The growing knowledge in Wikis is not accessible for machines; only humans are able to read and understand the knowledge in the Wiki pages while machines can only see a large number of pages that link to each other. This problem may negatively affect the Wiki's searching and navigation performance.

B. Semantic Wikis3

A semantic Wiki is a Wiki that has an underlying model of the knowledge described in its pages. While regular Wikis have only pages and hyperlinks, semantic Wikis allow identifying additional information about the pages (metadata) and their relations and making that information available in a formal language (annotations) such as Resource Description Framework (RDF) and Web Ontology Language (OWL) accessible to machines beyond mere navigation. Adding semantics (structure) to Wikis enhances their performance by adding the following features.

Contextual Presentation: Examples include displaying semantically related pages separately, displaying information derived from the underlying model of knowledge, and rendering the contents of a page in a different manner based on the context.

Improved Navigation: The semantic framework allows relating concepts to each other. These relations enhance navigation by giving easy access to relevant related information.

Semantic Search: Semantic Wikis support context sensitive search on the underlying knowledge base which allows more advanced queries.

Reasoning Support: Reasoning means deriving additional implied knowledge from the available facts using existing or user-defined rules in the underlying knowledge base. With these enhanced features, semantic Wikis can be valuable for software reuse. In addition to supporting general collaboration among users, semantic Wikis provide means of adding metadata about the concepts (artifacts) and relations that are contained within the Wiki. This system has the advantage of being easy to use for non-expert users while being powerful in the way in which new artifacts can be created and stored .

13. CONCLUSIONS

Software reuse is a longtime practiced method. Programmers have copied and pasted snippets of code since early days of programming. Even though it might speed up the development process, this "code snippet reuse" is very limited does not work for larger projects. The full benefit of software reuse can only be achieved by systematic reuse that is conducted formally as an integral part of the software development cycle. This paper gives a summary of some important aspects of software reuse research and presents a rough proposal for a software reuse repository system that is based on semantic wikis. The next step will be to further research the concept and implement a prototype to ensure its validity. In this paper, we have reviewed the history of software reliability engineering, the current trends and existing problems, and specific difficulties. Possible future directions and promising research problems in software reliability engineering have also been addressed. We have laid out the current and possible future trends for software reliability engineering in terms of meeting industry and customer needs.

14. REFERENCES

1. Curritt, P.A., Dyer,M, Mills, H.D, ACertifying the Reliability of Software,@ IEEE Transactions,
2. Software Engineering, vol SE- 12 no. 1 1994. Gert B (1988) *Morality*, Oxford University Press.
3. Green R M (1994) *The Ethical Manager*, Macmillan Publishing.
4. Gotterbam and Rogerson 1998, "The Ethics of Software Project Management", in *Ethics and Information Technology*, ed. G&an Collste, New Academic Publisher, 1998.
5. Humphrey, W. *A Discipline of Software Engineering* Addison Wesely Longman, Reading Mass, 1995.
6. Linger, R. ACleanroom Process Model,@ IEEE Software March 1994. pp 50-58.
7. Smith 1 9 9 C0. U] . Smith, *Performance Engineering of Software Systems*, Reading, MA, Addison-Wesley, 1990.
8. Smith and Williams 2002] C. U. Smith and L. G. Williams, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*, Boston, MA, Addison-Wesley, 2002.
9. Williams and Smith 2002a L. G. Williams and C. U. Smith, "PASASM: A Method for the Performance Assessment of Software Architectures," 2002 (submitted for publication).
10. Williams and Smith 2002Lb. G. Williams and C. U. Smith, "The Business Case for Software Performance Engineering," www.perfeng.com.