Volume 2, Issue 5, May 2012 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

An Emerging Scenario for Reusability of Software in Software Engineering

Ms. Mamta Gupta[1]
Research Scholar in Dept. of C.S.E.
S.V.I.T.S, Indore (M.P)
mamtaguptacs08@gmail.com

Prof.Dinesh Ch. Jain[2]
Reader, Dept. of C.S.E
S.V.I.T.S, Indore (M.P)
dineshwebsys@gmail.com

Abstract- ***In the field of software engineering the term reusability is the likelihood a segment of source code that can be used again to add new functionalities with slight or no modification. The concept of reusable (modules and classes) reduce the extra effort to implement again. This paper presents an empirical study of the software reuse activity by expert designers 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.***

Keyword: - SDLC, OOPS, Reusability of classes and modules.

I. INTRODUCTION

In the present scenario the software enhance 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 analyses 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 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.

II. 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, 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

III. RESUE –BASED SOFTWARE ENGINEERING

Application system reuse

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

Component reuse

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

Object and function reuse

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

IV. SOFTWARE RESUE BENEFITS

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

- **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 subsystems are reused.

- **Effective use of specialists**

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

- **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.

- **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.

V. RESUE PROBLEMS

- **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.

- **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.

- **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 software.

- **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
  .

- **Finding, understanding and adapting reusable components**

  Software components have to be discovered in a library, understood and, sometimes, adapted to work in a n new 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.

VI. 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.

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

- **Application product lines:-** An application type is generalized 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.

VII.PROGRAMMING LANGUAGES

The evolution of programming languages is tightly coupled with reuse in two important ways. First, programming languages have evolved to allow developers to use ever larger grained programming constructs, from ones and zeroes to assembly statements, subroutines, modules, classes, frameworks, etc. Second, programming languages have evolved to be closer to human language, more domain focused, and therefore easier to use. Languages such as Visual C++, Delphi, and Visual Basic clearly show the influence of software reuse research. The paper on the Fusion system by Weber et al. in this issue is a continuation of the trend of making large grained domain specific programming constructs, in this case business rules, available in a form closer to the language used by workers in the domain. Fusion also mixes declarative and algorithmic programming language approaches in a single system. Systematic reuse via domain engineering is another step in this direction. In systematic reuse, we consider how to codify and reuse subsystems and architectures. We attempt to establish the required vocabulary for a given problem area, apply it to the system building environment for that domain, and, thereby, build higher quality systems more productively. Reuse research has contributed to the widespread practice of design to interfaces, the practice of separating ff the shelf libraries of general components such as those for C, C++, Java, and C#. Some research on restricting the use of pointers in languages and better ways of handling reference aliasing has also been active.

VIII.MEASURING REUSE POTENTIAL

Before discussing our reuse guidelines, we need to define how to measure reuse potential of a component. In our work, reusability of a component is measured against the percent of reuse guidelines satisfied (i.e., reuse potential is the ratio of number of guidelines satisfied by a component and total number of guidelines that are applicable). This is similar to grading done for MCCR components (Gargaro and Pappas 1987). This grading system is essential when we assess a component for reuse and to report to reuse engineers about the component's strength for reuse. This can also help engineers to understand the further effort required to modify their components. Perhaps not all guidelines carry equal weight; therefore a significant factor (in terms of percentage) is attached to each guideline that is selected for automation.

A component may be:

1. Weakly reusable, whose potential for reuse is low which means it satisfies fewer than 50% of the guidelines. It needs more effort to redesign the original component for reuse.
2. Limitedly reusable, whose potential for reuse is high which means it satisfies between 50-70% of the relevant guidelines and needs some effort to improve it.
3. Strongly reusable, whose potential for reuse is high which means it satisfies between 70-90% of the relevant guidelines and needs little modification to improve it.
4. Immediately reusable, whose potential for reuse is very high which means it satisfies more than 90% of the relevant guidelines and this can be reused as-it-is without modification.

IX. 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". 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 a simple 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
- 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.

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 Wikis

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

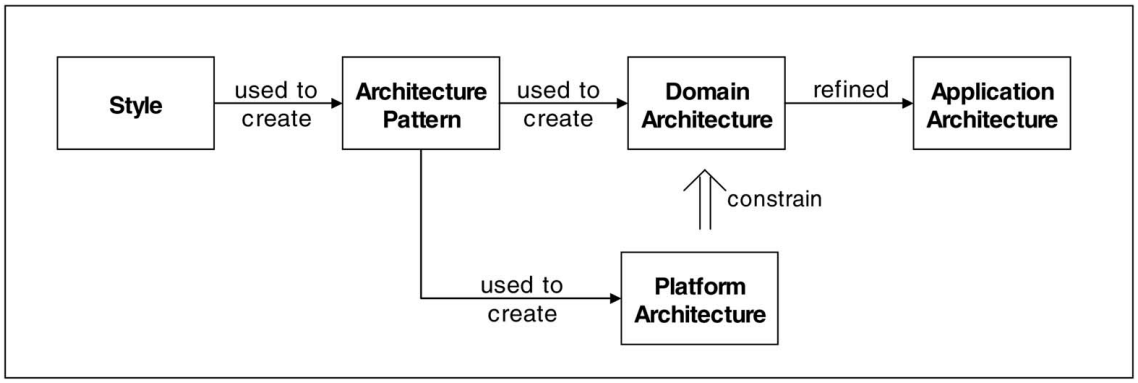

Fig1. Reusability of domain

X. 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.

XI. ACKNOWLEDGEMENT

Ms. Mamta Gupta is a research student (pursuing Me) in computer science & engineering from svits-Indore. Her research interest in the area of software engineering.

I would like to thanks my guide Prof. Dinesh Ch. Jain for supporting in this research work.

REFERENCES

1. Software Engineering, vol SE- 12 no. 1 1994. Gert B (1988) *Morality*, Oxford University Press.
2. Green R M (1994) *The Ethical Manager*, Macmillan Publishing.
3. Gotterbam and Rogerson 1998, "The Ethics of Software Project Management", in *Ethics and Information Technology*, ed. G&an Collste, New Academic Publisher, 1998.
4. 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.
5. Smith 1 9 9 C0. U]. Smith, *Performance Engineering of Software Systems*, Reading, MA, Addison-Wesley, 1990.
6. 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.
7. 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).
8. Williams and Smith 2002Lb. G. Williams and C. U. Smith, "The Business Case for Software Performance Engineering**,"** www.perfeng.com
9. C. Atkinson et al., Component-Based Product Line Engineering with UML. Addison-Wesley, 2002.
10.H. Gomaa, Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison-Wesley, 2004.
11. R. Ommering et al., "The Koala Component Model for Consumer Electronics Software," Computer, vol. 33, no. 3, pp. 78-85, Mar. 2000.
12. J. Kramer et al., "Software Architecture Description," Software Architecture for Product Families: Principles and Practice, M. Jazayeri et al., eds., pp. 31-64, 2000.
13. F. Buschmann et al., Pattern-Oriented Software Architecture. Chichester, UK; New York: Wiley, 1996.
14. W. Tracz, "DSSA (Domain-Specific Software Architecture) Pedagogical Example," ACM SIGSOFT Software Eng. Notes, vol. 20, no. 3, pp. 49-62, July 1995.
15. N. Leveson, Safeware: System Safety and Computers. Addison- Wesley, 1995.
16. B. Meyer, ".NET is Coming," Computer, vol. 34, no. 8, pp. 92-97,

17. *International Journal of Computer Applications (0975 – 8887) Volume 7– No.14, October 2010* 41.

