Software Reuse in Practice

Raman Keswani, Salil Joshi, Aman Jatain
Ramankeswani10itu052@itmindia.edu
Saliljoshi10itu061@itmindia.edu
Amanjatain@itmindia.edu
ITM University, Gurgaon -122017.

ABSTRACT

By software reusing we can expedite the development of a software product by re-using the components of another software product in a different behaviour. The concept of systematic software reuse is simple: the idea of building and using "software preferred parts." By building systems out of carefully designed, pre-tested components, one will save the cost of designing, writing and testing new code. The practice of reuse has not proven to be this simple however, and there are many misconceptions about how to implement and gain benefit from software reuse. This paper briefly summarizes software reuse research and discusses major research contributions.

1. INTRODUCTION

The component reused can be any part of its software development such as a code fragment, design, test cases, cost its software requirement specification. If any software developer needs to substantially improve the development, quality, efficiency and effectiveness then this reuse of components, process and knowledge it of pivotal importance.

Software reuse can be classified into two categories: deliberate reuse accidental and reuse. If a software particular is developed solely for reuse in future then it is deliberate reuse. On the contrary if a software developer comes to a conclusion that some previously developed software particular or component can be reused, then it is classified as accidental reuse.

Many of the previous studies on reuse guidelines such as by Dennis 1987; Braun and Goodenough 1985; and Booch 1987 although provided with many original software guidelines on software reuse which included issues such as documentation and interoperability, they were somehow unachievable.

In this paper, we will discuss about broad areas of development for reuse and how can we concoct achievable and objective guidelines for software reuse.

2. NEED FOR SOFTWARE REUSE

Reusing reduces efforts and save cost. Development costs of software can be very high but it's sharing. And reusing costs can be very low. One important benefit by reusing software is that it can significantly reduce the number of bugs. It is advisable to reuse proven legacy software rather than developing a product completely from scratch [3]. The goal of software reuse is to recycle the design, code and other components of a software product and thus reduce the cost, time and improve the quality of product.

3. MYTHS ABOUT SOFTWARE REUSE

Myth 1: Reusing Code Results in Huge Increases in Productivity.

There has been a lack of examples citing the reuse success stories. There really should be more success stories in the literature, but there aren't. While some instances show that reuse has resulted in a 20 to 1 reduction in effort for "certain" stages in the development life cycle, this should be placed in perspective with the "cost" of reuse [5]. There are primarily three costs associated with reuse: the cost of making something reusable, the cost of reusing it, and the cost of defining and implementing a reuse process. Focusing on the cost of making software reusable, a conservative breakdown is a follows:

25% for additional generalization
15% for additional documentation
10% for additional testing
5% for library support and maintenance

60% additional cost of making something reusable [5].

The subtle/sad thing is, reusability is quite subjective. It cannot be surely said that spending x% on documentation will make it y% more reusable, or an additional z% spent generalizing will make
it% more reusable1 Obviously, our aim is to not to spend money but to make certain attributes of the software better.

Myth 2: ADA has solved the Reuse Problem.

ADA has not, nor will it go away. Ada9X claims to have increased support for reusability in the form of an anaemic inheritance mechanism and object

978-1-4799-4910-6/14 $31.00 © 2014 IEEE
DOI 10.1109/ACCT.2014.57

types (but it still lends itself for improvement as well as abuse). As far as a language like C++ are concerned, they have become the standard languages for writing any new language despite their retro status but again, it is missing some parameterization capabilities that would enhance its applicability[5]

Myth 3: Software Reuse Will Just Happen
In five years time, Software reuse got a lot of push, but it really has not boomed. The DoD initiatives (e.g., STARS, ARPA's DSSA, and tile DISA CIM effort) are some instances and efforts where reuse has shown positive signs. The progress HP and IBM have made at institutionalizing software reuse as part of tile corporate programming strategy. [5]
The bottom line is that software reuse is maturing. It has evolved and in the next 5 years, One can expect it to bloom even more.

4. BARRIERS TO SOFTWARE REUSE

Why Software Reuse has Failed Historically?
Reuse has been a hot topic of debate and discussion for over 30 years in the software community. Developers have successfully applied reuse *opportunistically*, *e.g.*, by cutting and pasting code portions from existing programs into new programs [9]. Opportunistic reuse is a good way of implementing reuse when one is working individually or in small groups. However, it does not increase proportionally across business units or enterprises to provide *systematic* software reuse. Systematic software reuse is another way to reduce development cycle time and cost, improve software quality, and provide support to the existing effort by constructing and applying multi-use assets like architectures, patterns, components, and frameworks.

Like many other promising techniques in the history of software, however, systematic reuse of software did not boom the improvements in quality and productivity. There have certainly been successes though, *e.g.*, sophisticated frameworks of reusable components are now available in Object Oriented languages which run on many OS platforms. Generally, these frameworks have focused on a relatively small number of domains, such as GUI's or C++ container libraries like STL. Moreover, component reuse is often limited in practice to third-party libraries and tools, rather than being an integral part of an organization's software development processes.

Theoretically, organizations having recognized the value of systematic reuse, reward internal reuse efforts. In practice, however, many factors make systematic software reuse unfeasible, particularly in companies with a large installed base of legacy software and developers. In our opinion, non-technical barriers to successful reuse commonly include the following:

1. *Organizational barriers* -- *e.g.*, to reuse software one needs a deep understanding of application developer needs and business requirements only then one can develop and deploy old software for reuse [9]. The increase in the number of developers and projects employing reusable assets makes it becomes hard to structure an organization to provide effective feedback paths between these constituencies.
2. *Economic barriers* -- *e.g.*, supporting corporate-wide reusable assets demands economic investment, particularly if reuse groups need a huge investment [9]. Many organizations aren't able to institute appropriate taxation or charge-back schemes to fund their reuse groups.
3. *Administrative impediments* -- *e.g.*, Owing to the large size of industry it becomes very difficult to reuse software or part of it outside one's workgroup as an organization has multi business units, so docketing and archiving reuse across multiple business units becomes infeasible [9].
4. *Political impediments* -- *e.g.*, application developers are often suspicious about the group that develops reusable middleware platforms, application developers always harbour a grudge that the reusable middleware platform developers they may no longer be empowered to make key architectural decisions. So the internal conflict between different groups which poses a threat to the job security causes the reuse of software components difficult. [9]
5. *Psychological impediments* -- *e.g.*, Application developers may not want a "top down" approach as it indicates that the management does not have faith in their technical skills, highly talented programmers are also against reuse as they believe in developing everything their way and reuse causes a not made her kind of an attitude.

As if these non-technical impediments aren't daunting enough, reuse efforts also frequently fail because developers lack technical skills and organizations lack core competencies necessary to create and/or integrate reusable components systematically. For instance, developers often lack knowledge of, and experience with, fundamental design patterns in their domain, which makes it hard for them to understand how to create and/or reuse frameworks and components effectively.

We observed that developers commonly focus on language features, such as inheritance, polymorphism, exceptional handling, and templates for software reusability. But sadly, the language features are not alone sufficient to capture common components for efficient reuse of software

5. REUSE GUIDELINES

- **Create and maintain a central library**

  A software development institution should have central library storage. The programmers creating new components from scratch or from their previously developed projects must place them in central library making them readily available for reuse in future [7].

- Document the submissions

  The submissions into the central library must be very well organized and proper guidelines must be established for these submissions. Create documentation or a cover which clearly specifies attributes such as date of creation, code (or design) functionality, its usage in business process etc[6]. The documentation should be submitted along with its test cases so that developer reusing this component can verify it easily and quickly. It must also include guidelines of how to apply the component within the software product. Also they must be easily accessible for all developers of that organisation.

  By reusing software to shorten the critical path in delivering a product, organisations within HP have reported a reduction in time-to-market from 12% to 42%. Improve software quality
  Using any software over time reflects many bugs which were not discernible when it was created. So a software product that has been reused many times will contain much less bugs and defects than freshly created software. An organisation of HP which had seasoned reused code had a ten times better defect density than its new code. Quality improvements in HP products from reuse have ranged from 24% to 76%.

- Change The Definition Of Success

  It is imperative that a company understands and encourage the importance of reuse within the organisation. Those developers effectively reusing software must be rewarded appropriately.[7]

- **Use Code Reviews**

  Software cannot be reused at each development phase and during each functionality addition of an application product [2]. To reuse software a developer must understand a situation by using code reviews to know if reusability is feasible or not.

- **Perform Cost Analysis**

  A software development organisation must perform analysis so as to determine reusing software will result in cost reduction or not. However cost analysis must not be the only criteria for the final decision.

  Cost can be analysed by:

  Net cost saving = Cost of development from scratch - continuing costs associated with reuse – actual cost of the software [8]

- **Human Resources**

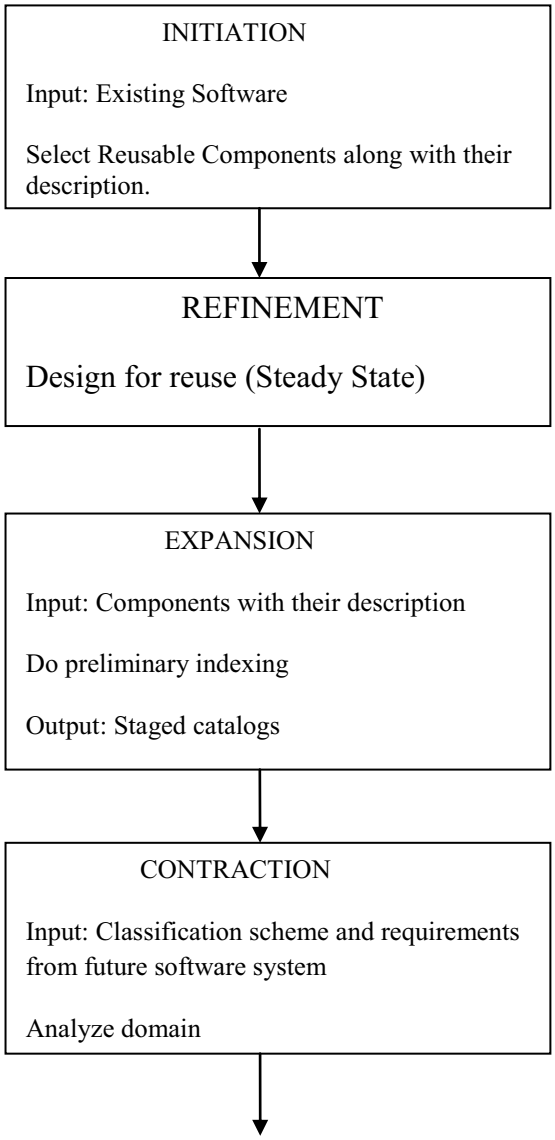
  Due to shortage of proper checks on reuse processes, reusing sometimes turns out to be a big failure. To make it absolutely successful on organisation must understand the need for an experienced developer to manage its developers and their responsibilities. As a higher authority they must assign roles and keep a check on working of their subordinates. Like components being added into databases and identifying new components required.

- As a thumb rule, every important software development activity should provide 2 to 3 new components, and, once the library is big enough (100-200 components) less than half the code of the new software should be developed particularly for that new software

- **Minimize Risk**

  Another important guideline for software reuse is to reduce the risk in creating new software. It can be achieved if the components being reused have the needed features and interfaces such that they can be easily integrated.

6. SOFTWARE REUSE ADOPTION MODEL



The above figure shows a proposed adoption model for software reuse. A reuse program can be implemented in four basic stages: Initiation, Expansion, Contraction, and Steady State.
Redundant components are removed. This step must be performed by domain analyst experts such as software engineers.

- ***Stage 1: Initiation*** - In this stage potential reusable components are identified from existing software. Component descriptors are extracted

- ***Stage 2: Expansion*** - As more reusable components are identified, catalogue size increases

- ***Stage 3: Contraction***- In this stage, domain analysis is carried out. Common components are identified and are integrated [1].

- ***Stage 4:*** *Steady State*- Once essential components are indentified (for a specific domain), they are replaced as domain specific functions [1]. These components are explicitly designed to be reusable.

CONCLUSION

A reuse program to give appropriate returns must be systematic and planned. Any organization implementing software reuse should identify best reuse methods and strategies for maximum productivity. Software reuse helps in avoiding engineering software from scratch as it uses existing software modules. It is possible to reuse software components if they can be re-engineered appropriately. Software reuse although a difficult task especially for legacy software, can significantly improve productivity and quality of a software product. Although software reuse is not a new field, it can give large returns in short period of time.

ACKNOWLEDGMENT

We take this opportunity to express our profound gratitude and deep regards to our guide Ms Aman Jatain for her exemplary guidance, monitoring and constant encouragement throughout the research work.

REFERENCES

[1] Rubén Prieto-Díaz, “Making Software Reuse Work: An Implementation Model” :Software Productivity Consortium.
[2] M. Ramachandran: “ Software Reuse Guidelines” : Liverpool John Moores University
[3] Yijun Yu, (2005) “Software Reuse” [4McGrawHill, schachh8(online).
[5] Kevin D. Wentzel, “Software Reuse - Facts and Myths”: Hewlett-Packard Laboratories
[6] Ali bahrami (1997) “Object oriented System Development “
[7] http://www.arnull.co.uk/ , Reuse Guidelines, 2002.
[8] Nasib S. Gill. “Reusability Issues in Component-Based Development” M.D. University, Rohtak
[9] Douglas C. Schmidt:1999 “Why Software Reuse has Failed and How to Make It Work for You”
[10] William Frakes, Carol Terry “Software Reuse: Metrics and Models”, Virginia Tech
[11] William B. Frakes and Kyo Kang, “Software Reuse Research: Status and Future” , 2005
[12] Ruben Prieto-Diaz , “Implementing Faceted Classification for Software Reuse ‘, 1991ss