Designing A Framework for Handling Barriers to Software Reuse

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Software Engineering

Submitted By
Jyotsna
(801031014)

Under the supervision of:
Ms. Shivani Goel
(Assistant Professor)

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004
June 2012

CERTIFICATE

I hereby certify that the work which is being presented in the thesis entitled, "*Designing A Framework for Handling Barriers to Software Reuse*", in partial fulfillment of the requirements for the award of degree of Master of Engineering in Software Engineering submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of Ms. Shivani Goel and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

Jyotsna

(Jyotsna)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

Shivani Goel.

(Ms. Shivani Goel)
Assistant Professor,
Computer Science and Engineering Department
Thapar University
Patiala

Countersigned by

(Dr. Maninder Singh) Head Computer Science and Engineering Department Thapar University Patiala	**(Dr. S. K. Mohapatra)** Dean (Academic Affairs) Thapar University Patiala

Acknowledgement

I would like to express my sincere gratitude to all who have made possible the fulfillment of this work.

Firstly, I would like to thank mu guide, Ms. Shivani Goel, Assistant Professor, CSED, Thapar University, Patiala for the time, patience, guidance and invaluable advises she has given me not only while my thesis work but throughout the course. It was a great opportunity to work under her supervision.

Then I would like to thank Dr. Maninder Singh, Head of the Department, CSED, Thapar University, Patiala for providing all the facilities and environment. I would also like to thank all my teachers for their support and invaluable suggestions during the period of my work.

I would also like to thank my parents for always supporting me in the tough and happy moments, for their never ending support and inspiration.

Finally, I wish to thank to my brother, Ashish Kumar for being with me through the good and the bad.

Jyotsna

Jyotsna

(801031014)

Abstract

Software reuse has received much attention since the concept was introduced in 1968. Software reuse itself is a broad concept that has many levels of meaning, ranging from strict code reuse, to design/analysis reuse, to entire application reuse. Reusable software components are designed to apply the power and benefit of reusable, interchangeable parts from other industries to the field of software construction. Component-based Software Engineering is an approach to software development that relies on software reuse. Benefits of component reuse are sharing common code, making components available at one place and making development easier and quicker. Many companies, especially those in the defence and aerospace fields, have setup successful reuse programs. However, not all reuse programs are successful. There are many barriers to software reuse which impede the successful implementation of software reuse in software development. The barriers can be broadly categorized into three aspects namely, technical aspect, managerial/organizational aspect and cultural aspect. Technical barriers can be considered from technical point of view like lack of tools, lack of technologies, lack of methodologies etc. Managerial barriers are related to management in the organization like lack of management support, lack of incentives. Any barrier which is related to any change in the organization or related to processes will be related to cultural barriers.

This thesis identifies all the barriers to software reuse, the reason behind these barriers and provides the solution for the barriers.

Abbreviations

CBSE	Component Based Software Engineering
FODA	Feature Oriented Domain Analysis
ODM	Organization Domain Modeling
JODA	Joint Object-Oriented Domain Modeling
NIH	Not Invented Here
CBSD	Component Based Software Development
CMM	Capability Maturity Model
AOP	Aspect Oriented Programming
OOP	Object Oriented Programming
ROI	Return On Investment
RAIS	Reuse Assessor and Improver System
SMART	Service Migration and Reuse Techniques
CASE	Computer Aided Software Engineering
CAPE	Computer Aided Protocol Engineering
ST	Software Thesaurus
COM	Component Object Model
AUTOSAR	Automative Open system Architecture
CCM	Corba Component Model
EJB	Enterprise JavaBeans
OMG	Object Management Group
PECOS	Pervasive Component System
PCM	Palladio Component Model
SOFA	Software Appliances

Table of Contents

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

This chapter introduces a description of work presented in thesis. It gives a brief introduction of software reuse, component based software engineering(CBSE), the assessment for software reuse and the benefits of software reuse.

1.1 Background

Despite several decades of intensive research, demand for developing new ever more complex software systems and for maintaining existing software keep on rising and also the production of software under acceptable conditions of productivity and quality remains an unfulfilled promise. However reuse can have a significant and largely positive effect on software development and provide some remedies to the current software crisis[1].

Software reuse was first proposed by McIlroy at a NATO workshop in 1968. Since that time, software reuse gained much attention and has taken its place in the array of software engineering and also being implemented in industries[2]. Among all the software engineering concentration areas, reuse becomes the centre of attraction perhaps being the easiest to understand, while being the hardest to implement successfully. There are some documented software reuse success stories which shows the successful implementation of reuse and there are also documented accounts of how reuse has failed to meet expectations due to certain barriers.

1.2 Software Reuse

Krueger's general view of software reuse:-

"*Software reuse is the process of creating software systems from existing software rather than building them from scratch. Software reuse is still an emerging discipline. It appears in many different forms from horizontal reuse and vertical reuse to systematic reuse, and from white box reuse to black-box reuse*"[3]. In software systems, many people misunderstands software reuse as the reuse of software code alone as it is mostly reused. Source code and design reuse have become popular with class libraries, application frameworks and design

patterns. Many different viewpoints exist of what software reuse is. For Freeman “Reuse is the use of any information which a developer may need in the software creation process”[4]. Similarly Basili and Rombach think “Software reuse as the use of everything associated with a software project, including knowledge”[5]. Braun defines reuse as "The use of existing software components in a new context, either elsewhere in the same system or in another system”[6]. An important aspect is whether software to be reused may be modified. For Tracz “Reuse is the use of software that was designed for reuse”[7]. Cooper defines “Software reuse as the capability of a previously developed software component to be used again or used repeatedly, in part or in its entirety, with or without modification”[8]. Lim mentions “Work products (i.e., products or by-products of the software development process, e.g., code, design, test plans) that are to be used in the development of other software without modification”[9]. Additionally, we affirm McIlroy's vision of reuse: the goal is the use of off-the-shelf components as building blocks in new systems with modifications occurring in a controlled way. It is not always possible to simply reuse components. Development of components or systems is still necessary[10].

Figure 1.1 defines the difference between current and future models for software reuse.

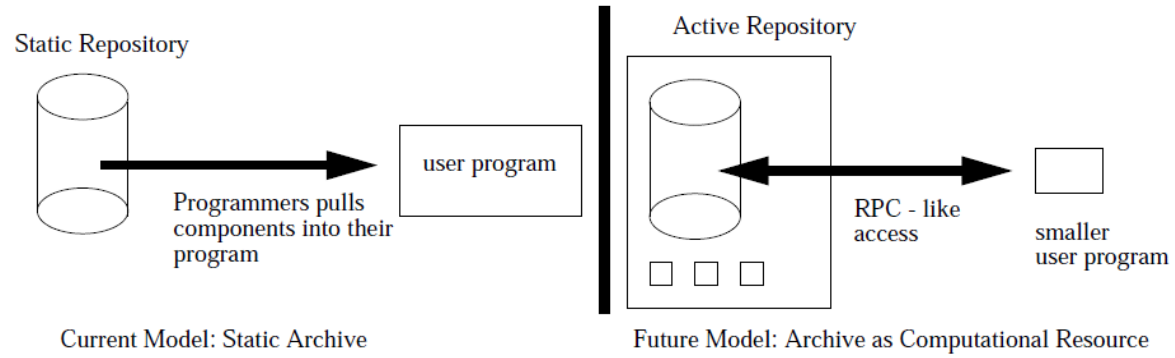


Figure 1.1: Current and Future Models of Software Reuse[2]

1.3 Need to reuse software

Reuse is being considered to solve three main problems in software: high cost, low quality and low productivity. Reuse also increases the reliability and decreases the implementation time. So software reuse results in improvements in quality, productivity, performance, reliability and interoperability. An initial investment is high to start a software reuse process,

but the investment pays for itself in few reuses. Reuse of software components has been taken from manufacturing industry and civil engineering field. Examples are manufacturing of vehicles from parts and construction of buildings from bricks[11]. Spare parts should be available in markets to make it successful. Software companies have used the same concept to develop software in parts[11]. The software parts are called components. A component is an independent part of the system having complete functionalities.

Four levels of reuse are proposed:

1. code level components (modules, procedures, subroutines, libraries, etc.)

2. entire applications

3. design level products

4. analysis level products

Code level component reuse occurs most frequently. Examples of this are standard libraries and popular language extensions. However, the level of abstraction is low for these components and that places a limit on the amount of reuse that can be expected. Reusing entire applications, with little to no modification, is great when it can happen, but is just not feasible for many real world problem domains. Using entire applications often means using off-the-shelf packages e.g., Microsoft Office[2]. Figure 1.2 shows different forms of reuse. Reuse can take any of these forms.

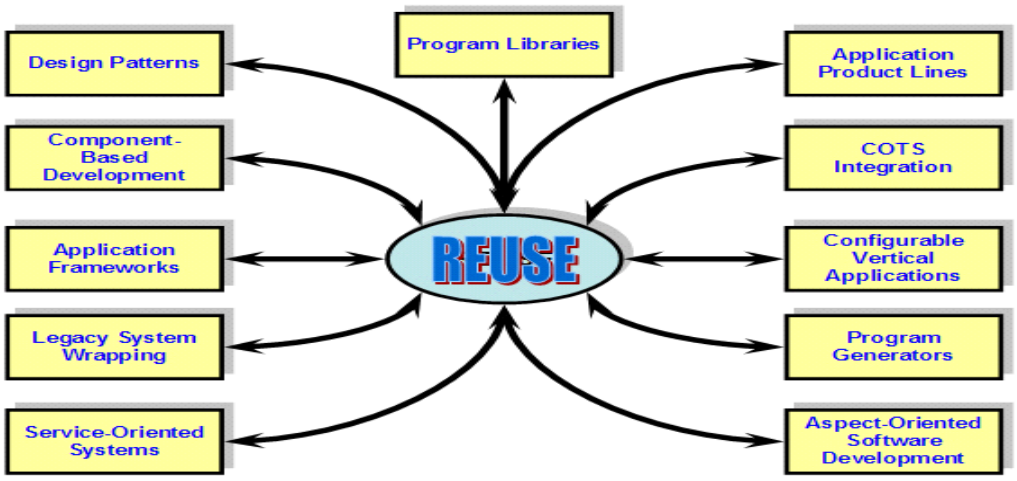


Figure 1.2: Different forms of software reuse

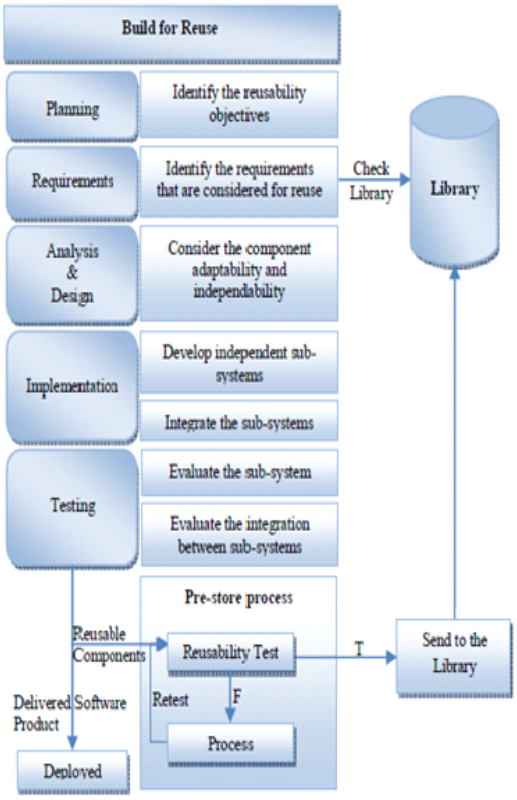
1.4 Assessment of Reuse

While there are articles documenting efficiencies and savings gained, the general perception within the computing community and those that fund it, is that software reuse has yet to fully deliver on its promises of higher quality software produced quicker and for less money. There are many structural reasons offered for the partial success of reuse, including: researchers not focusing on reuse methods for a wider range of applications, analysis, structural, and architectural mismatch of target reuse components, and organizational alignments that are not optimum for fostering reuse. This is not to imply that there has been no progress in software, but rather that software reuse is different from many realms of computer science. Reuse involves the standard amount of technical challenges, but it also has significant non-technical problems that are not easily resolved[2].

1.5 Component-Based Development

Component Based Software Engineering(CBSE) is an emerging software engineering paradigm in which applications are developed by integrating existing components. The purpose of CBSE is to develop large systems, incorporating previously developed or existing components, thus cutting down on development time and costs. It can also be used to reduce maintenance associated with the upgrading of large systems Here, components refer to any units of reuse or integration, including computational components, interface components, communication components, and architectures[12]. The component based software engineering process is quite different from that of the traditional waterfall approach. CBSE not only requires focus on system specification and development, but also requires additional consideration for overall system context, individual components properties and component acquisition and integration process[13]. Component-based software engineering is a special case of software engineering for and with reusable assets: reusable assets have special packaging and are referred to as components[14]. To develop software by integrating components, components must be developed for reuse. Therefore, CBSE must address both the development of reusable components and the development applications using the reusable components**.** Component based development shows the perspective of software reuse.

The link between develop for reuse and develop by reuse is very important in the success of the new project development It supports both frameworks shown in figure 1.3 and figure 1.4.



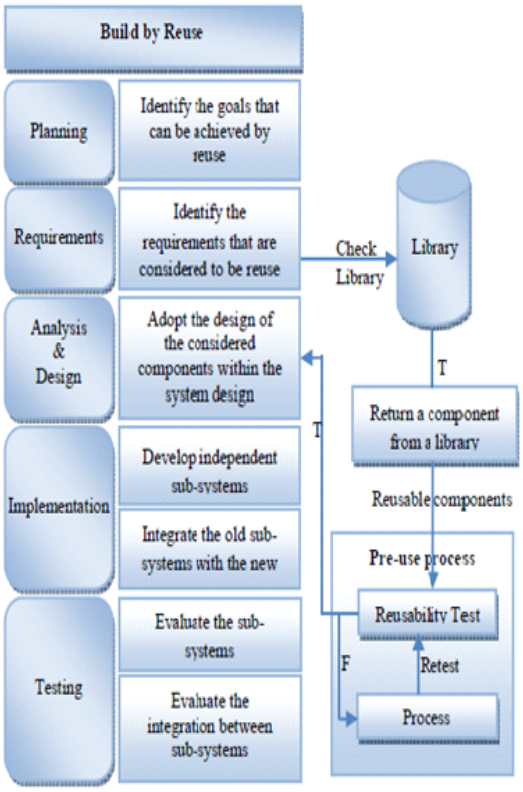


Figure 1.3:Build-for-reuse Framework **Figure 1.4: Build-by-reuse Framework[15]**

1.5.1 Activities in component-based development

In component-based development, the components are to be present in software repository for reuse. Activities involved are:-

Find:- The process of finding components defines how to document and create repositories of components. Finding a component is an activity in the domain engineering phase.

Select:- We select components from the repository in order to use them in component-based development. The selection process is usually related to library retrieval and browsing techniques and algorithms.

Adapt:- Adaption is a process of customizing selected components to satisfy user requirements in the new context in which component is used.

Create:- In component-based development, it is sometimes possible that the selected components do not fully satisfy application requirements after adaption. In such case, the product integrator has to develop and create new components for the specific application.

Compose:- Composition is assembly-and-integration process. The effort of integration depends on the nature of the components to be integrated.

Replace:- The replacement process is related to product maintenance. Component-based systems evolve over time to fix errors in components and add new functionalities[14].

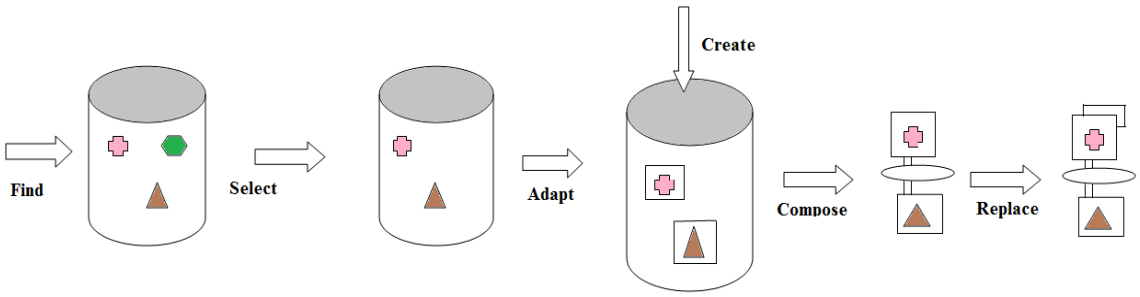


Figure 1.5: Activities in component-based development[14]

1.6 Benefits of Software Reuse

Software requires a cost for construction and maintenance of software repository. This money once spends can reap a number of benefits which are discussed here:-

a) Higher Quality

Error fixes accumulate from reuse to reuse i.e. with more and more reuse the errors in a component are corrected and eliminated. So the reused component yields higher quality for a reused component than the component that is developed and used only once. However, this requires the administration and maintenance of components and is not achieved by simply reusing[2,10,11,16].

b) Enhanced Productivity

Productivity is defined as the number of projects developed in a unit time. A productivity gain is achieved when less code has to be developed. This results in less testing efforts and also saves analysis and design labour and saves the overall cost[2,10,11]. When reuse is being installed, productivity may decrease shortly due to increased learning effort and the need to develop reusable components. This temporary decrease in productivity should easily be compensated by long-term increase in productivity due to reuse[15,16].

c) Better Performance

Software reuse increases the performance as extensive reuse can be worth the effort invested in optimizations. Performance of a reused is better than the component that is developed and used only once. However, generalizations that make components more reusable can have a negative influence on overall performance. For example, for Avionics simulator software, Bardo et al. have reported 15-20% penalty in load module size with minimum overhead execution[10].

d) Increased Reliability

"Reliability is the probability of a device performing its purpose adequately for the period of time intended under the operating conditions encountered"[17]. Well tested components increases the reliability of a software system. The reuse of a component in several systems increases the chance of errors to be detected and strengthens confidence in that component[10,11,15,16].

e) Interoperability

Interoperability can defined as the ability of two or more entities to communicate and cooperate regardless of differences in the implementation language, the execution environment, or the model abstraction[18]. Various systems can work better together if their interfaces are implemented consistently. This is the case when they use the same components for these interfaces. Even though written standards improve interoperability, different implementations might differently interpret parts of these standards[10,19].

f) Reduction in Redundant work, development time

When every system in being developed from scratch, redundant development of many parts like user interfaces, communication, basic algorithms, etc increases. This can be avoided when these parts are available as reusable components and can be shared, resulting in less development time and costs[10,11,15,16].

g) Shorter Time to market

The success or failure of a software product is very often determined by its time to market and product should be available in market as early as possible due to market competition. Using reusable components will result in a reduction of that time. A specific project has been considered and 42% reduction has been reported using reusable components[10.15].

h) Less documentation

Documentation is considered as important for the maintenance of a system and still it is often

neglected. Reusing software components reduces the amount of documentation to be written but stresses the importance of what is written. Overall structure of the software system and the newly developed components have to be documented. The documentation of reusable components can be shared among many software systems[10].

i) Low Maintenance costs

Reusable components are already tested. So fewer defects are expected to occur when proven components have been used, and less of the software system must be maintained. The reusable components are maintained by a separate group rather than separately in each software system[10,11,15,16].

j) Low Training costs

Software engineers become familiar with the reusable components available for development with span of time. So engineers have a good working knowledge and experience of many components of these systems when they are starting to design and develop new systems. Software reuse leads to low training cost.[10,11,15].

k) Team size

Doubling the size of a development team does not result in doubled productivity as large development teams suffer from a communication overload. Team also depends on the reusability of the components, if many components can be reused, then software systems can be developed with smaller teams, leading to better communication and increased productivity[10].

l) Rapid prototyping support

We can quickly build a prototype of a software system using these reusable components. This also helps in providing the opportunity to get customer feedback early in the life cycle, thus supporting the conception of the requirements[10].

m) Expert sharing

It is important that software engineers study the designs of excellent peers in order to improve their design skills as good design can only be learned from good designers. Software reuse supports this very naturally. There is no need to study the implementation details of all the components we reuse, the interfaces alone are sufficient to know the important information about how a component and its interoperability have been designed[10].

1.7 Industry Examples

Reuse benefits have been reported in various industrial settings.

An empirical study from NASA software production environment has shown that modules reused without modifications had less interaction with human users, and higher ratios of commentary compared to new developed or modified modules. In this study 25 software projects were considered. An average of 32% of software had been reused or modified from previous systems.

•At Motorola software reuse is considered a candidate technology for initiatives and goals to improve productivity and quality.

•At Hewlett-Packard a reuse assessment of two reuse programs has indicated higher quality(reduction in defect ranging from 24% to 76%) and a 40% to 57% increase in productivity.

•Raytheon Missile Systems has reported an average of 60% reuse and a 50% increase in net productivity in new developments.

•Universal Defense Systems has reported 60% reuse in a system of 700,000 lines of Ada code[10].

1.8 Organization of thesis

Chapter 2 defines different aspects of software reuse as software reuse is broadly categorized into three aspects.

Chapter 3 includes the problem statement and the scope of thesis work

Chapter 4 includes the identification of the barriers which influence the success of the software reuse. It also identifies the critical barriers.

Chapter 5 provides the framework for handling the barriers which includes the reason behind those barriers and also includes the solution for those barriers.

Chapter 6 provides the detailed solution to the most critical barriers.

CHAPTER 2

ASPECTS OF SOFTWARE REUSE

Previous chapter gave a brief introduction about software reuse, its need, benefits and assessment. It also gave the overview of component based software development. This chapter discusses the different aspects of software reuse.

2.1 Different Aspects of software reuse

Software engineering has been focused on original development but it is now identified that to achieve software quickly and at lower cost, we need to adopt a design process that is based on systematic software reuse. Software reuse is multidisciplinary and can be broadly categorized into two aspects: technical and non-technical aspects. The non-technical aspects can be further categorized into organizational and cultural aspect. So the software reuse has three main aspects: technical aspects, organizational aspects, cultural aspects.

2.1.1 Technical Aspects

Technical aspects of software reuse can be divided into three classes: application engineering aspects, domain engineering aspects and component engineering aspects.

a) Application Engineering Aspects

Application engineering is concerned with the development of application using reusable assets. The development approach can be top-down or bottom up. In the reuse-based top down approach for application development, the first phase is requirements specification where application's requirements are being specified. Next phase is product design, where the system's design is refined so that it is possible to identify design units when found in the reuse library. Then is retrieval, where the units are searched in the reuse library and assessed to determine whether they fulfil the design specification. After that, coding of the units that were not found in library is done and then integration and testing. On the other hand, reuse-based bottom-up approach is different where reuse library is visited before the design. It means first step is requirements specification, then browse the library and then proceed to develop the design. Application engineering with reusable assets involves the different steps.

like asset retrieval and assessment, asset adaptation and asset composition[14].

Browsing:- Browsing is performed before the product design and is the process of navigating the software library.

Retrieval:- Retrieval consists of navigating the library in order to find assets that satisfy the specifies requirements and is performed after the product design. Retrieval methods can be of two types: exact retrieval and approximate retrieval. In the exact retrieval we seek to identify the asset from library which exactly fulfils the requirements. While in approximate retrieval we seek to identify asset that almost fulfil all the requirements.

Assessment:- Assessment depends on the type of the retrieval. If it is exact retrieval, assets are evaluated with respect to the query to select that which provides the best fit and is it is approximate retrieval, assets are evaluated with respect to the query to select that which minimizes the modification effort.

Adaptation:- Adaptation is also categorized into two types: blackbox reuse and whitebox reuse. In blackbox reuse, reusable assets are integrated without modification and in whitebox reuse, reusable asset are being analyzed and modified before these get integrated. Exact retrieval fits in the lifecycle of blackbox reuse and approximate retrieval fits in whitebox reuse.

Composition:- Composition depends on the nature of the reusable software assets. If the reusable assets are finished software products in executable form then the compositional development is applicable but if reusable assets are represented in terms of patterns then generative development is applicable.

b) Domain Engineering Aspects

Domain engineering is concerned with the development of reusable asset across an entire application domain. So domain engineering team is responsible for producing, maintaining and cataloguing reusable assets and to make them available to application engineering teams. Domain engineering involves different tasks like domain analysis, domain engineering, asset acquisition, asset classification and asset maintenance[14].

Domain Analysis:- Domain analysis is the process of capturing, analyzing and modelling information about application in the domain. Domain analysis helps in scoping the domain and also checks whether it is worth to develop a reuse infrastructure for the domain. It also identifies the commonalities and variabilites that exist among application of domain. Domain analysis is accomplished by reengineering technique and domain analysis methods like feature-oriented domain analysis(FODA), organization domain modelling(ODM), joint object-oriented domain modelling(JODA), synthesis domain analysis method etc.

Domain Engineering:- After domain analysis, domain engineering team builds the reuse infrastructure that is required to support application development within the domain.

Asset Acquisition:- This phase consists of development for reuse with some procurement of reusable assets. Development for reuse is different from traditional development due to the additional features needed to meet the requirement of genericity, reliability and maintainability.

Asset Classification:- It consist of cataloguing and storing reusable asset to facilitate their retrieval and assessment by application engineering team.

Asset Maintenance:- This involves the maintenance tasks which get merged with a configuration management/version control task

Domain engineering is the process of building reusable assets to help application engineering achieve its goal. The figure shows the relation between domain engineering and application engineering

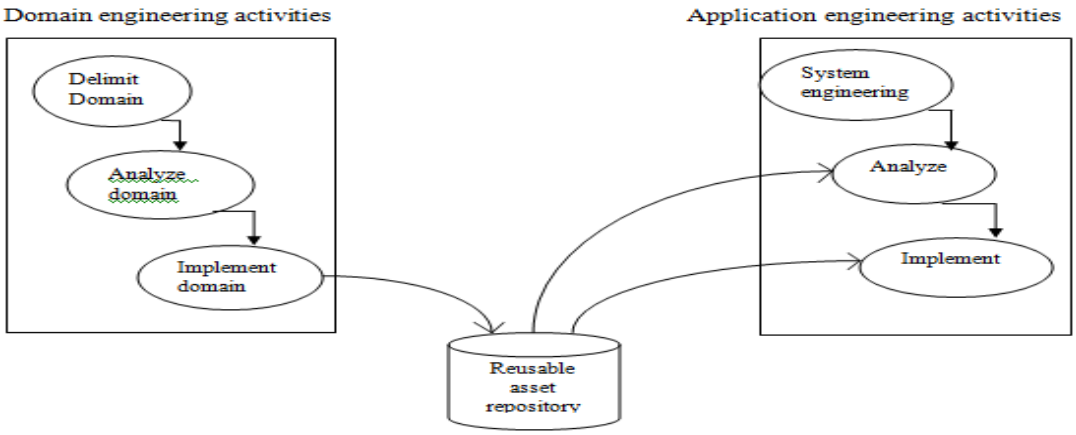


Figure 2.1: Producer-consumer model of domain engineering[14]

c) Component Engineering Aspects

Component Engineering deals with how to develop reusable assets. Reusability of asset depend of two features: usefulness and usability.

- Usability is a domain engineering issue and refers to how little it cost to reuse this asset. Modularity is an important ingredient for usability as it emphasizes simple interface specifications and low coupling.

- Usefulness is a component engineering issue and refers to how often the asset is expected to be reused. Genericity is important for usefulness as it enhances the frequency of reuse of an asset by enabling the user to tailor the asset to specific needs.

2.1.2 Organizational Aspects

At the organizational level, two sets of measures are required for the smooth operation of a software reuse initiative: a managerial infrastructure and a technological infrastructure as these structures involve nontrivial changes.

a) Reuse Introduction

Reuse introduction is basically concerned with introducing systematic reuse in an organization with little or no reuse experience. There are certain problems which have been identified in the reuse introduction process. The problem is based on many factors like high initial cost for developing, lack of management commitment, lack of training for reuse etc. For reuse to get initiated, the management must be convinced of possible gains. The top level managers should be informed about the benefits of reuse and also about the problem to avoid unrealistic expectations[20]. Software reuse in software development organizations is accompanied by nontrivial changes, so it must be carefully planned and executed. It is usually dependent on a reuse champion, who secures the cooperation of all the parties involved to bring about the necessary organizational changes. Reuse champion should have clearly specified goals and must be able to measure progress towards these goals. They should also know the risk involved and their impact. Reuse introduction is dependent on precise plan of action, and on managerial support. Reuse introduction is also dependent on the willingness of asset developers to work within the domain engineering paradigm, and the willingness of

application developers to give reusable asset consideration. The stepwise introduction of reuse involves the recruitment of staff, evolution of development processes and procedures for measurement[14]. Figure 2.2 shows the process of reuse introduction[20].

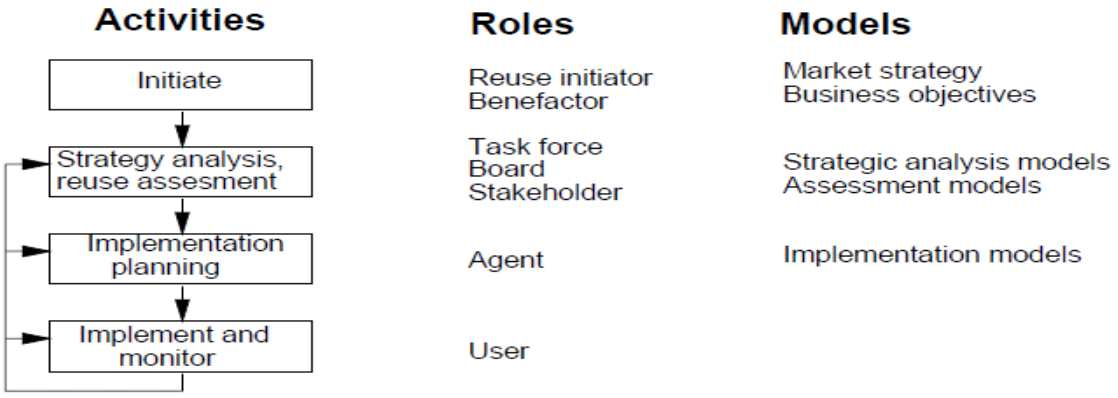


Figure 2.2 Reuse Introduction Process[20]

b) Managerial Infrastructure

Reuse is not just a technical thing to understand. Management support and adequate organizational structure is equally important. The managerial infrastructure is the set of functions, responsibilities, reporting requirements, and reward of incentive mechanism that are required to ensure the operation of reuse processes. The organizational measures for software reuse must be merged with existing measures for traditional software development. Software reuse program cannot be accomplished without the support of management. In order to have successful implementation of reuse, managers must be convinced about the importance of reuse program[14]. There are certain barriers like lack of management support, lack of incentives, turf battles etc which impede the successful implementation of software reuse[21].

c) Technological Infrastructure

The technological infrastructure includes all the technical functions that must be provided to support reuse operations. The main function is the library support function which include configuration management function, as various versions of the same product are used in more and more applications and the configuration management function keep track all the versions in existence. Quality assurance(QA) function must be supported along with the configuration

management function as quality assurance process ensures that the software development process and software products ensures to established standards. These standards apply to reusable asset level and application level. There are testing standards: verification and validation standards and certification standards that must be established and enforced at the asset level and the application level. Software verification and validation methods are used to increase the level of assurance of critical software. Finally standards for risk assessment and risk analysis must be established and enforced by the technological infrastructure[14].

2.1.3 Cultural Aspects

There is another area that needs adequate attention is the corporate culture which must be supportive of reuse initiatives and efforts. Any change to an organization or its processes will face resistance as it is easier to do the things the way they are done now. Software reuse technique also require certain changes in traditional software development. So there are many cultural barriers which software developer has to face while adopting the software reuse. The is number of attitudes like fear of unknown, apathy, not invented here(NIH) syndrome, fear of failure etc that can resist implementing software reuse within the organization[21].

CHAPTER 3

PROBLEM STATEMENT

In previous chapter we explored different aspects of software reuse. We have also seen the advantage of using software reuse. Despite the benefits of software reuse, there are many factors that directly or indirectly influence the success or failure of software reuse. Software reuse in the broad sense has many level of meaning ranging from code reuse to entire application reuse. Many companies have setup successful reuse programs but not all the reuse programs are successful. As reuse tries to solve three main problem: low quality, high cost and poor productivity. With the advantages of software reuse, there are many barriers of software reuse which impede the successful implementation of software reuse in software development. There are several technical issues that currently keep reusable software from becoming a reality. The barriers can be technical barriers or non-technical barriers. Non-technical barriers are further classified as managerial and cultural barriers. So the barriers can be broadly categorized into three aspects namely, technical aspect, managerial/organizational aspect and cultural aspect. Technical barriers can be considered from technical point of view like lack of tools, lack of technologies, lack of methodologies etc. Managerial barriers are related to management in the organization like lack of management support, lack of incentives. Any barrier which is related to any change in the organization or related to processes will be related to cultural barriers. Earlier it was thought that the technical barriers are the main barriers which influence software reuse but later on it was concluded that non-technical barriers are equally important. Due to these barriers, software reuse is still an emerging technology. So the barriers must be resolved in order to implement software reuse in the organization.

The aim of the thesis work is to identify all the barriers to software reuse, to identify the reason behind the barriers and also to provide the solution for the barrier. So the research objectives of this thesis are:-

1.To identify the barriers to software reuse.

2. To identify the critical barriers.

3. To identify the reason behind the barriers to software reuse.

4. To propose a framework to provide ways to overcome the barriers.

CHAPTER 4

RESEARCH FINDINGS

As discussed in the previous chapter, there are many factors which influence the success of software. In this chapter all those barriers are identified so that solutions can be provided to overcome those barriers.

4.1 Barriers to Software Reuse

A list of barriers that influence the success of software reuse are identified from literature review and has been categorized in three categories as shown in Figure 4.1:

S.No	Technical Barriers		Managerial/Organizational Barriers		Cultural Barriers
1	Structure Mismatch		Lack of top management support		Apathy
2	Steep learning curve		Lack of management incentives		Not invented here syndrome
3	Poor cataloguing, Distribution & access methods		Infrastructure clash		Fear of the Unknown
4	Modification		Turf Battles		Ivory Towerism
5	Integration		Never enough time and money		Unfamiliarity with reuse tools and techniques
6	Garbage Reuse		Quality issue	Reliability	Education and Ego
				Security	
7	Lack of Tools and Technologies		Legal issue	Data Right	
				Copyright	
8	Lack of Methodologies		Inadequate organizational structures		
9	Object Oriented Technology	Doesn't allow strong implementation reuse	Cost of reusing and making thing reusable		
		Tangling of code			
		Low level Abstraction			
10	Lack of explicit procedures		Liability		
11	Non-reusability of found software		Difficult in reuse measurement		
12	Component Qualification				
13.	Legacy components not suitable for reuse				

Figure 4.1 Barriers to Software Reuse

Software reuse needs a cost but if succeeds can give many benefits. However it is not always easy to achieve successful software reuse. There are many barriers which make software reuse difficult.

1. Structure Mismatch

Most software development organizations(SDOs) have embraced paradigms, methods, and tools that do not promote the use of software components. Instead of attempting to build systems using collection of reusable components, many SDOs assume everything must be designed and developed anew. They argue that their user requirements drive them towards such unique solutions. Incorporating component concepts into such methods is difficult because the SDOs do not provide support for the types of trade-offs needed to assess the performance impacts of components. In addition, component-based development(CBD) is further complicated by the lack of tools needed to build systems using templates, software component models, software component infrastructures and building block libraries. Example for structure mismatch is when input and output format does not match[21,22].

2. Steep learning curves

Due to conceptual mismatches of CBD and the current development practices, severe learning curves for component reuse exists. Those trying to exploit the use of components in software systems often have to open their minds to embrace new ways of doing business. The developer need to know the new ways of doing business but this would take many months to become proficient and competent in CBD[21,22].

3. Poor Cataloguing, Distribution and Access Methods

Software cannot be reused unless it can be found. Reuse is not likely to happen when the components are poorly classified or when a repository does not have sufficient information about the components. Suppose we know that there exists a software component that exactly matches our needs but finding it is impossible unless we have a well-organised repository containing the particular component with some means of accessing it[2,10,11].

4. Modification

Components will not be always available exactly in the way we want them. If modification is necessary, we should be able to modify them and able to determine their effects on the

components. The amount of modification required affects the time and money required for white box reuse[10].

5. Integration

Components are available with the functionality that is needed for a new software system. If it is not possible to integrate components into the system, they are of no use. Unfortunately most components are ill-equipped to cope with integration requirements. Software components must be constructed in a way that subsequent reuse can be efficient and straightforward[10].

6. Garbage reuse

Poor quality control is one of the major barriers to reuse. Certifying reusable components to certain quality levels help in minimizing possible defects. A reusable component should perform the functions for which it is designed. We need some means of judging whether the required functions match the function that are provided by a component or not[10].

7. Lack of tools and technologies

Lack of tools and technologies is one of the critical barriers identified. Tools and technologies are not sufficient enough to support reuse. Software reuse required proper tools and new technologies to support reuse efficiently[10,16].

8. Lack of Methodologies

The lack of methodologies is one of the main barrier which influence the success of software reuse. There should be proper methodologies available in order to enhance reuse. For example, object oriented technologies support reuse but it has certain limitations which impede the proper implementation of reuse[16].

9. Object Oriented Technology

It is believed that object-oriented technology has a positive influence on software reuse. Unfortunately and wrongly, many also believe that reuse depends on this technology or the adopting object-oriented technology suffices for software reuse[10]. This is due to the following:

a) Doesn’t allow strong implementation of reuse:- As each module implements several concerns, so it is not able to support reuse completely. In this individual module is less loosely coupled as compared to other programming techniques[23].

b) Tangling of code:- This problem occur with object-oriented programming as certain software properties cannot be isolated in a single function and they cross cut multiple components which results in tangling of code[23,24].

c) Low level abstraction:- This occurs when OOP does not address each concern separately with minimum coupling[23].

10. Lack of explicit procedures

Software development involves many activities and software reuse has much effect on the whole software lifecycle like design, methods, project planning and estimating. Models for such processes use to determine various steps to be accomplished in a certain order. If these models do not explicitly consider software reuse, it is not possible for software reuse to happen in practice[10].

11. Non-reusability of found software

Ease access to software does not necessarily increase software reuse. Unintentionally software is seldom written in a way so that others can reuse it. Modifying and adapting someone else’s software can prove to be expensive rather than programming the needed functionality from scratch[10].

12. Component Qualification

Component qualification can depend on the rating system. A rating system for code reuse would make people more comfortable reusing software. However, this would be a large task and leads the suspicion and questions to another level: who has assigned this rating? Who has authorized this rating system? Such systems are generally possible within the limited scope of an individual company[11,16,25].

13. Legacy components not suitable for reuse

It is not possible to reuse components unless they have been designed and developed for

reuse. Simply gathering existing components from various legacy software systems and trying to reuse them for new developments is not sufficient for systematic reuse. Re-engineering can help in extracting reusable components from legacy systems[10,14].

14. Lack of top management support

Software reuse requires initial funding, organizational restructuring, new processes and revised business practices which cannot be widely achieved in an organisation without support of top-level management. Managers have to be informed about initial cost and have to be convinced about expected savings[2,10,16,14].

15. Lack of management incentives

Mangers are not given proper incentives. So the lack of incentives prohibits managers from letting their developers spend time in making components of a system reusable. Their success is often measured only in the time needed for completing the project. They will not work beyond the completion of project although beneficial for organization[2,10,11,16,26].

16. Infrastructure clash

Mostly large organization have already established the software management infrastructure consisting of capability maturity model(CMM)- compatible process and decision rules. The possibility exist that infrastructure may need to change to introduce new technology such as software components. New technology introductions tend to bring more distractions because they impact the manner in which decisions are made and work is accomplished[21,22].

17. Turf battles

When responsibilities and budgets for the components are being assigned, turf battles generally occurs. Malignity may occur when organization compete for tasking budget and power. Some bitterness seems to be stay long even after turf battles appears to be settled. There should be some ways to sort this problem[21,22].

18. Never have enough time and money to do things right

This problem is somehow closely related to turf battle barrier. There is inadequate time, people and money to do things right Adequate resource need to be made available otherwise

possibility exists that whomsoever senior management offers the authority and responsibility for implementing CBD will fail to deploy the technology throughout the organization[21,22].

19. Quality issues

These are issues in evaluating the quality of components and how the quality attributes of an application can be assessed as components are integrated. These issues arises while developing with reusable assets.

- **Reliability:-** Ensuring the reliability of software application is a difficult task, even when pretested and trusted software components are integrated together to develop the application[14].

- **Security:-** There are several issues related to overall security of an application defined in terms of the security of individual components. There is issue of modelling component security and to preserve the security of overall application while integrating these components. So the main problem is how to prevent the security-broken components from compromising the overall application security[14].

20. Legal issues

The more widespread software reuse becomes, the more legal and business issues have to be addressed e.g copyright and data rights. Increased used of third-party software increases the significance of these issues[2,10,14,27].

21. Inadequate organizational structure

Organizational structures must consider different needs that arise when explicit, large scale reuse is adopted. For example, separate teams may be installed for gathering, maintaining and providing reusable components[10].

22. Cost of reusing and making things reusable

Reuse initial cost is very high, it can save money in the long run. Developing components for reuse is more expensive than developing them for single use only. Cost of reusing the component is also unpredictable as there is lack of economic model to do the cost-benefit analysis[10,16].

23. Liability

Reuse is slowed by producers not wishing to be liable for their components performance in other system, and by the consumers of reusable components not wishing to risk their system on potentially unknown software[2,10,11,16].

24. Difficulty in reuse measurement

We know that we cannot manage what we cannot measure. Software reuse also follows the same rule. Reuse spans multiple projects and has an influence even on organizational structures of companies. To manage such activities there required some kind of monitoring. Software metrices can be used to estimate costs, cost saving and value of the software[10].

25. Apathy

Apathy is a problem related to adoption of tools and methods. It is not a new problem. Any change to an organization or its processes will face resistance, since it is easier to just continue to do things the way they are done now. "why change when nothing is broken" is a question change agents are bound to hear[14,21,22].

26. Not invented here syndrome

People feel hindered in their creativity and independence by reusing someone else's software. They want to develop their own new software rather than maintaining others software. There is lack of trust in their mind on someone else's software. "Just because it works elsewhere doesn't mean it will work here" is the comment software components advocates must address[14,21].

27. Fear of unknown

People are sometimes fearful of the unknown. In this "Why should we institute change now?" is the common question component and reuse advocates are likely to hear. People sometimes wonder why they should consider the new technology[14,21].

28. Ivory Towerism

Ivory towerism sometimes may feel its impact when trying encourage CBD adoption as some developers consider CBD as immature technology. Reuse deals with emerging technologies, many people argue that it might be premature to try to use something like components now.

Many critics argue that immature technology drains resources from other projects[14,21].

29. Unfamiliarity with reuse tools and techniques

People are unaware of existing reuse tools and techniques and don’t use these tools and this factor influence the success of software reuse[14].

30. Education and Ego

Software reuse is not taught in academic training. Computer science departments make it quite clear to students that reuse and cheating are the same thing. Yet, upon arrival at the job, the same people are expected to undergo a radical transformation and understand how to program in teams. This initial educational bias often manifests itself later in programmer's careers as the suspicion of “if I reuse it, others will think I’m not smart enough to write it myself.”[2,11].

4.2 Research approach

Literature survey was done in order to identify the various barriers for software reuse. Many researchers have identified a large number of barriers to software reuse. In order to relate the barriers, three broad categories were identified for grouping various barriers. The stakeholders for software reuse are the organization, top management, developers and customers. In order to survey for the critical barriers, a questionnaire was used as a tool. The questionnaire was divided into three major categories viz. technical, cultural and managerial. The technical section contained ten technical barriers. The managerial section had ten barriers. Five barriers were included for survey in cultural section. The questionnaire is attached as Appendix-A.

The users were asked to give the input to some of the questions on a 3-point Likert scale with values as 1-Not important, 2-important, 3-Very important. Some of the questions required the prioritization among the given options. The questionnaire was sent to 50 software developers, but complete response was collected from 20 only. The users included users from industry and students doing projects with software reuse.

4.3 Research findings

(a) The list of barriers has been divided into three categories: technical, managerial and cultural.

Technical Barriers:- The barriers which are considered from technical point of view are technical barriers and categorization of technical barriers involves: Structure Mismatch for reuse, Steep learning curve with reuse, Poor cataloguing, distribution and access methods, modification, integration, lack of tools and technologies, lack of methodologies, object-oriented technology(doesn't allow strong implementation reuse, tangling of code, low level abstraction), lack of explicit procedures.

Managerial Barriers:- Managerial barriers assumed to be non-technical type of barrier and this managerial barriers involve: Lack of top management support, lack of management incentives, infrastructure clash, turf battles, never enough time and money to do things right,

quality issue(reliability, security), legal issue(data right, copyright), inadequate organizational structures, difficult in measurement of reuse, cost of reuse.

Cultural Barriers:- Cultural barriers are also assumed to be non-technical type of barriers and this cultural barriers involves: Apathy, not invented here(NIH) syndrome, fear of the unknown, ivory towerism, unfamiliarity with reuse tools and techniques.

(b) Critical technical barriers: The results of the survey for identifying the critical technical barriers are summarized in table 4.1:

Table 4.1: Average Score for Technical Barriers

S.No.	Technical Barriers	Average Score
01	Structure Mismatch in reuse	2.05
02	Steep learning curve with reuse	2.05
03	Poor cataloguing, Distribution & access methods	2.25
04	Modification for reuse	2.00
05	Integration during reuse	2.05
06	Garbage reuse	1.90
07	Lack of tools and technologies for reuse	2.95
08	Lack of Methodologies for reuse	2.40
09-1	OOP doesn't allow strong implementation reuse	2.30
09-2	OOP causes tangling of code	2.75
09-3	OOP has low level of abstraction	2.30
10	Lack of explicit procedures for reuse	1.95

From the average score, the lack of tools and technologies is identified as the most critical

implementing software reuse. The tangling of code caused by object oriented technology is the next most critical technical barrier. Lack of methodologies , lower level of abstraction by object oriented programming and poor cataloguing, distribution and access methods are identified as critical barriers for software reuse.

Figure 4.2 shows the chart for the technical barriers. It involves the technical barriers as explained earlier. This chart is plotted on the basis of the average score defined in table 4.1

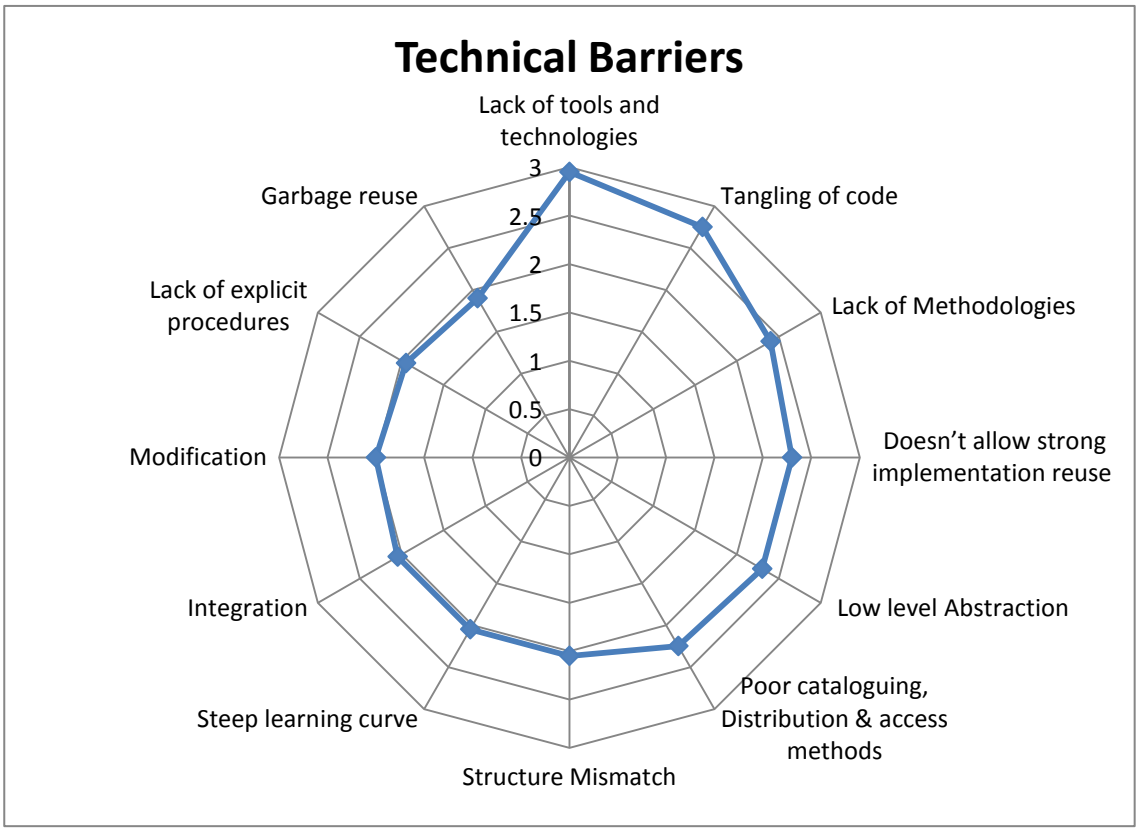


Figure 4.2 Chart for Technical Barriers

(c) Critical managerial barriers: The next category for barriers in software reuse is managerial or organizational level. The cost for reuse initiative, implementation, training and maintenance of software repository is to be handled by the top management in the organization. So managerial barriers have been identified and we will try to calculate the average score for those barriers using the questionnaire and then on the basis of those average score, critical barriers will be identified.

Various barriers identified were surveyed for identifying the critical barriers for management. The results of the survey for identifying the critical managerial barriers are summarized in the table 4.2.

Table 4.2: Average Score for Managerial Barriers

S.No.	Managerial Barriers	Average Score
01	Lack of top management support	2.60
02	Lack of management incentives	2.50
03	Infrastructure clash	2.20
04	Turf Battles	1.70
05	Never enough time and money	1.95
06-1	Reliability	2.40
06-2	Security	2.50
07-1	Data Rights	2.60
07-2	Copyright	2.50
08	Inadequate organizational structures	1.95
09	Difficulty in measurement of reuse	2.40
10	Cost of reuse	2.40

The lack of top management support and issue of data rights in reuse are identified as the most critical barriers for management in software reuse. Lack of management incentives for promoting reuse and security issues in reusing the software developed by other developers are also identified as critical barriers. The other critical barriers identified for management are reliability, difficulty in measurement of reuse and cost of reuse.

Figure 4.3 shows the chart for the managerial barriers. This chart is plotted on the basis of the average score of the managerial barriers defined in table 4.2

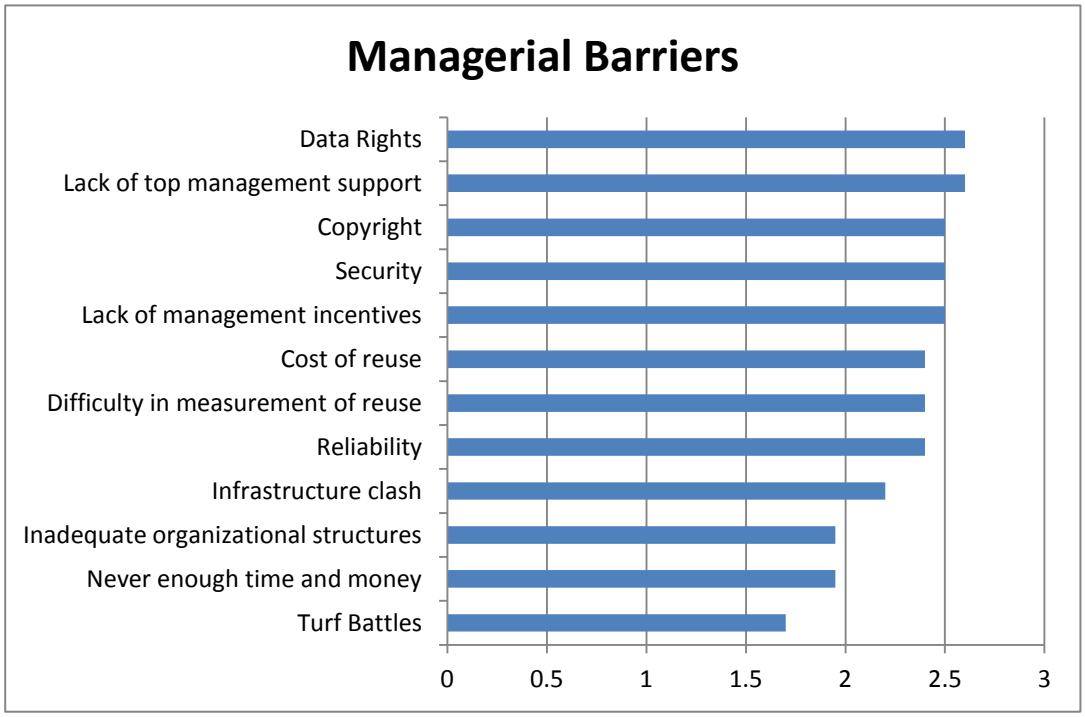


Figure 4.3 Chart for Managerial Barriers

(d) Critical cultural barriers: When we want to institutionalize the reuse, there are many hurdles from cultural point of view also. This is because the mind set of the software developers need to be changed in support for software reuse. There are many cultural barriers identified by many researchers. Five of them were included in the survey. The results of the survey for identifying the critical cultural barriers are summarized in the table 4.3:

Table 4.3: Average Score for Cultural Barriers

S.No	Cultural Barriers	Average Score
01	Apathy	2.00
02	Not invented here syndrome	2.50
03	Fear of the Unknown	2.30
04	Ivory Towerism	2.10
05	Unfamiliarity with reuse tools and techniques	2.30

The Not Invented Here (NIH) syndrome is identified as the most critical barrier in software reuse. The next critical barrier for software reuse is the fear of the unknown. Most users fear about the security and reliability of the software to be reused. Also there is found an unfamiliarity with reuse tools and techniques which is a critical barriers even for the users who want to develop with reuse and for reuse.

Figure 4.4 shows the chart for the cultural barriers. This chart is plotted on the basis of the average score of the cultural barriers defined in table 4.3

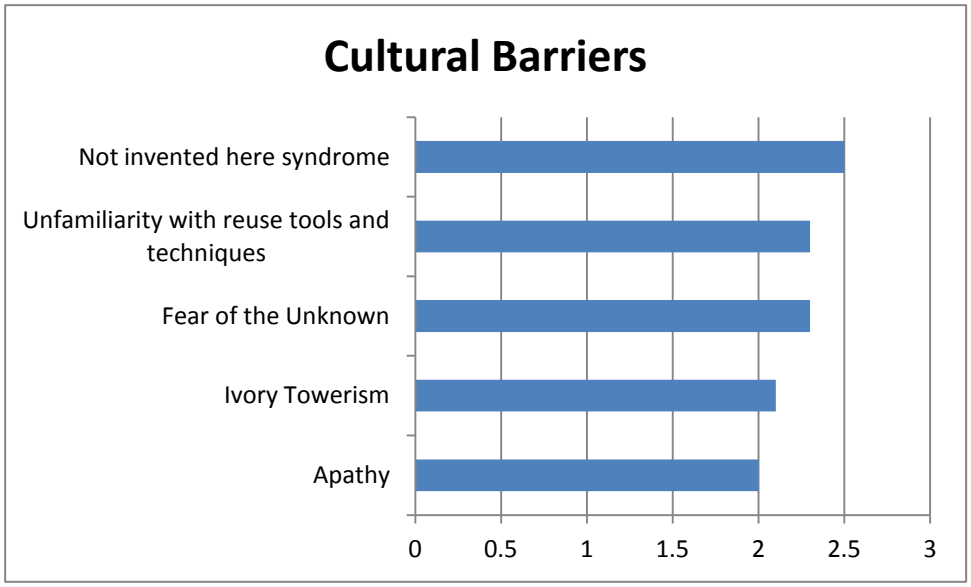


Figure 4.4 Chart for Cultural Barriers

After identifying the critical barriers in these categories, a framework is proposed to handle these barriers. All the barriers identified here from the literature survey are important. So the solutions to all the identified barriers are given in the framework shown in chapter 5.

If we compare the most critical technical barriers , the following barriers are identified as lack of tools and techniques ,tangling of code and lack of methodologies supporting reuse. The detailed solutions to the most critical technical barriers are given in chapter 6.

CHAPTER 5

SOLUTION TO BARRIERS

In the previous chapter we identified the barriers which influence success of software reuse. We also identified the critical barriers for software reuse. In this chapter, we will provide provides the framework for handling the barriers which includes the reason behind those barriers and also includes the solution for those barriers.

S.N	Technical Barriers	Reason	Solution
1	Structure Mismatch	Less support for types of trade-offs	Guidelines, tools and prototype is being needed
2	Steep learning curve	Conceptual mismatches	Training and mentoring
3	Poor cataloguing, Distribution & access methods	Repository are badly organized & less means to access it	Access hierarchy and access tools required
4	Modification	When components are not according to requirement	Abstract virtual machine interface to be developed
5	Integration	Components are ill-equipped	Component should satisfy integration requirements
6	Garbage reuse	Poor quality control	Quality assurance & testing
7	Lack of tools and technologies	Technologies supporting reuse still not developed	Various component model
8	Lack of methodologies	Methodologies supporting reuse still not developed	oop,aop,sop,vop
9	Object-Oriented Technology		
9.1	Doesn't allow strong implementation reuse	Module is less loosely coupled	Individual module is more loosely coupled due to AOP
9.2	Tangling of code	OOP crosscut multiple component	Separation of concern in AOP
9.3	Low level Abstraction	Module implements several concerns	AOP addresses each concern separately with min coupling
10	Lack of explicit procedures	Models don't consider reuse	Model should include reuse
	Managerial Barriers	**Reason**	**Solution**
1	Lack of top management support	Long term goal of project reuse and initial investment is high	Managers should be convinced about benefits of reuse
2	Lack of management incentives	To have financial savings	Incentives, rewards be given
3	Infrastructure clash	New technology introduction	Careful transition is needed
4	Turf Battles	When responsibilities and budget are being assigned	Everyone should be part of discussion
5	Never enough time & money to do right	Lack of adequate resources like time, talent, people and dollars	Schedule expectation to be set & resources to be available
6	Quality issue		
6.1	Reliability	components used not tested properly	Estimation techniques being used
6.2	Security	Risk in component design, integration & unauthorized resources & access	Constraints on components and security framework is needed
7	Legal issue		
7.1	Data Rights	When there is increased used of third-party software	Legal constraints should be removed or reduces if feasible
7.2	Copyright		
8	Inadequate Organizational structure	When explicit, large-scale reuse is being adopted	Separate teams need to be installed
9	Cost of reusing & making reusable	Developing components for reuse is more expensive	Economic model is needed
10	Difficult in reuse measurement	Lack of ways to measure	Metrices like reuse level,ROI & reuse maturity can be calculated
	Cultural Barriers	**Reason**	**Solution**
1	Apathy	When change is made to organization or processes	Planned business plan needed
2	Not invented here syndrome	Lack of trust on someone else's software	Use success stories and positive result of pilot project
3	Fear of the unknown	Emergence of new technology	Convince senior & middle mgmt
4	Ivory Towerism	While dealing with emerging technologies	Provide proof of successful implementation of technology
5	Unfamiliarity with reuse tools and techniques	Short term goals of completing project at hand & doing it in traditional way	Training ,awareness and benefits of reuse should be defined

Figure 5.1: Framework for handling barriers to software reuse

5.1 Solutions To Barriers

1. Structure Mismatch

Solution:- To overcome the mismatch between general software development and CBD, guidelines need to be developed and CBD tools are required. Examples also need to be developed to serve as prototype of what is expected when the modified paradigm, methods and tools are used to generate products[21,22].

2. Steep learning curve

Solution:- As the developers should take keen interest in new ways of doing business , it would take several months to get experienced for that. So the integration of training and mentoring on smaller, less mission-critical CBD and reuse projects is generally suggested and prove to be less risky[21,22].

3. Poor cataloguing, Distribution and access methods

Solution:- Access hierarchy for repository like NASA Technical report server should be developed. Tools should be properly designed for designers to locate reusable software in the repositories. For example, Code finder, a retrieval tool which helps in retrieving software components when information needs are not properly defined and users are not familiar with vocabulary used in repository[28,29].

4. Modification

Solution:- Abstract virtual machine interface should be developed in order to overcome this problem up to some extent. A component should provide a complete and minimum interface. The component must also hide implementation decisions of the component so that different implementations can be made to components[10,12].

5. Integration

Solution:- Sometimes component possess the functionality which is required but still it is not possible to interface component with new components. So components should be constructed in way so that its reuse is efficient and simple[10].

6. Garbage reuse

Solution:- Quality assurance of components, proper documentation standards used and rigorous testing should be done in order to overcome reuse and to enhance software reuse for its effective working[10].

7. Lack of Tools and Technologies

Solution:- This is one of the most critical barriers identified and should be provided with solutions to overcome this problem. Different tools like RAIS, ARCSeeker, SMART, Software Thesaurus and different component model like koala, rubus, com, Kobra, Autosar, Fractal, EJB should be used to enhance software reuse[10,16].

8. Lack of Methodologies

Solution:- This is also one of the most critical barriers identified. There are different methodologies like object oriented technology, aspect oriented programming, subject oriented and view oriented programming have been developed in order to handle this barrier and improve software reuse.

9. Object-Oriented Technology

- **Doesn't allow strong implementation reuse**

Solution:- Object-oriented Technology doesn't allow strong implementation reuse, more code reuse is possible using aspect-oriented programming(AOP) as it implements each aspect as a separate module and individual module is more loosely coupled[23].

- **Tangling of Code**

Solution:- This is one of the main problem while using object-oriented technology but the separation of concerns in AOP solve this problem[23].

- **Low Level of abstraction**

Solution:- AOP has high level of abstraction as compared to object-oriented programming as AOP addresses each concern separately with minimum coupling.. So better understanding is possible using AOP[30].

10. Lack of explicit procedures

Solution:- Software development process consists of many activities. These models should consider software reuse in every phase. So the process models including software reuse should be developed and used in order to enhance software reuse[10].

11. Lack of top management support

Solution:- Management support for the software reuse is the main key ingredient for the successful implementation of software reuse. Management should know the benefits of reuse. Managers have to be informed about initial costs and have to be convinced about expected savings[10,14,20].

12. Lack of management incentives

Solution:- Lack of incentives prohibit managers and their developers to spend more time in making the component reusable. So the project managers and developers should be given proper incentives, rewards, tokens and gifts to enhance software reuse[14].

13. Infrastructure clash

Solution:- The existing infrastructure need to be change in order to facilitate introduction of CBD. Careful transition is needed in order to succeed in such a cultural change in and to overcome this problem[21,22].

14. Turf Battles

Solution:- Proper care must be taken to ensure that there is consensus over who does what for inserting components in the organization and all should be the part of the discussion so that chances of this problem get reduced[21,22].

15. Never have enough time and money to do things right

Solution:- Proper allocation of resources is important. For this, schedule expectations, investment & budget appropriated need to be made based upon small, prototypical projects and proper mix of resources need to be available. This way the problem can be reduced and hence the adoption of CBD[21,22].

16. Quality issues

- **Reliability**

Solution:- Several techniques are there to estimate and analyze the reliability of application developed. The techniques are System-Level Reliability Estimation and Component-Based Reliability Estimation where the system-level estimation is estimated for the application as a whole and component-level estimation is estimated using the reliabilities of individual components and their interconnection[14].

- **Security**

Solution:- Components should be developed with strict constraint on security such as firewalls or encryption software and a security framework of component based system security is needed[14].

17. Legal issues

Solution:- Legal and contractual constraints on reuse are removed or reduced to increase the potential for reuse when feasible[14].

18. Inadequate Organizational structure

Solution:- To overcome this problem, a separate team may be installed for gathering, maintaining and providing reusable components[10].

19. Cost of making thing reusable and cost of reusing it

Solution:- As initial cost for reuse is high and there is now way exists to perform the cost-benefit analysis for reuse. New economic model should be developed and used to explain the benefits and cost of software reuse[14,31].

20. Reuse metrices aspect

Solution:- We cannot manage the thing which we cannot measure. Software metrices can be used to estimate costs, cost saving and value of the software. Four metrices namely, reuse level, lines and words run, Return On Investment(ROI) and reuse maturity can be useful to overcome this problem[10,31]. These metrices can be measured in the following ways:-

(a) Reuse level:- Amount of software reuse can be determined by ratio of reused components to the total components of the system.

Reuse level = number of reused / total number of components

(b) Lines and words Run:- Line reuse percentage is ratio of number of identical lines to total number of line multiplied by 100.

Lines and words Run = (Number of identical lines / Total number of lines)*100

(c) Return on Investment:- Return on investment is the ration of reuse savings to the generalization costs.

Return on Investment = Reuse savings / Generalization costs

(d) Reuse Maturity:- Reuse maturity can be seen as the range of expected result of reuse efficiency, reuse proficiency and reuse effectiveness.

21. Apathy

Solution:- Software reuse and CBD proponent can counter the response by building technical and business case for components and handling the cultural issues related to change. A well planned, published business plan can help to overcome this problem and to show critics that it is better to make changes[21,22].

22. Not invented here syndrome

Solution:- Use success stories and recommend manager to visit successful software component implementation project that they can relate easily also use positive result of pilot project and use the theme "see components work here too"[21,22].

23. Fear of the unknown

Solution:- Convince senior and middle management as well as key designers & developers that organisation is ready for transformation. They must be persuaded to embrace change and be able to appreciate the potential benefits of this initiative[21,22].

24. Ivory Towerism

Solution:- Provide proof of successful implementation of technology at variety of large and small development Organization. Also made confirmation from project managers at some of companies that have successfully implemented CBD and CBSE[21,22].

25. Unfamiliarity with reuse tools and techniques

Solution:- Training and awareness about tools, techniques and benefits of reuse should be provided[14]. Use of training package should be there.

Training package[20] involves three courses:-

Managerial Briefing. This is a two hour briefing for managers, explaining the possible benefits of reuse, as well as the most important problems. The mission of this course is to increase managers awareness of reuse.

Reuse Beginners Course. This is a course for developers who have little knowledge about software reuse principles.

Advanced Reuse Course. This is a course for more advanced developers, who have already been through the Beginners Course, or otherwise acquired knowledge about software reuse.

CHAPTER 6

SOLUTION TO CRITICAL BARRIERS

In the previous chapter ,the framework for handling the barriers was provided. We are able to provide the reason and solution for the barriers. This chapter will emphasize on the critical barriers. This detailed solution to the most critical barriers is being defined in this chapter.

Solution to Critical Barriers

From the analysis done in chapter 4, some of the barriers like lack of tools and technologies, tangling of code and lack of methodologies supporting reuse has been identified as most critical barriers. Here we will provide solution to those most critical barriers.

6.1 Tools

In order to overcome the lack of tools barrier, a list of tool has been defined to facilitate software reuse. The tools supporting software reuse are:-

(a) Reuse Assessor and Improver System (RAIS)

Reuse Assessor and Improver System is prototype tool which can identify, analyze, assess, and modify abstractions, attributes and architectures that support reuse. This tool provide automated improvement for component reuse as it takes the existing components, provides systematic reuse assessment which is based on reuse advice and analysis, and produces components that are improved for reuse[32]. RAIS is a system that supports the design for reuse process by automating the process of language-oriented and domain-oriented reuse assessment and improvement[33]. Reuse assessment is a process of assessing the reuse potential of a component based on the number of reuse guidelines that are satisfied by the component. Reuse improvement is a stepwise process of improving a component for reuse through several transformations. Reuse improvement transforms an assessed component into a component that is improved for reuse, based on language-oriented and domain-oriented reuse guidelines[32].

(b) ARCSeeker

ARCSeeker is a tool to manage reusable asset and support reuse by collecting scattered information and also improves the retrieval performance[34]. It is the utility tool which supports to manage and reuse Sparx System Enterprise Architect's UML models. We can create reusable components from Enterprise Architect's UML models using ARCSeeker and other information like source files, documentation files can be added for the components. It means we can gather various software assets such as UML models, documents and source files, put them together and store them as a component[35]. The tool also mentions how to search and use the stored information. It is difficult to manage many UML models and to search the component we want to reuse, so better is to divide the model into small units to be referred and used easily and efficiently.

(c) Service Migration and Reuse Technique(SMART)

Service-oriented architecture (SOA) is a software architectural paradigm which is a collection of independent, self-contained services that can be accessed in a standard way and SOA environment make best use of legacy systems by recasting existing capabilities as services. SMART is a technique to help organizations make initial decisions about the feasibility of reusing legacy components as services within an SOA environment. SMART tool supports information gathering and analysis activities of SMART and also produces draft migration strategy and migration issues list. The tool was developed mainly in Java. The tool has two major components— SMART Client and the SMART Server[36]. The SMART Client is a Java application built using the Eclipse Rich Client Platform (RCP). This run on the laptop of a SMART facilitator in offline mode during the engagement. The SMART Server is a Web application that runs on a central server of an organization that performs SMART engagements.

(d) Computer Aided Software Engineering(CASE) Tools

CASE tool is one of the tools that support software reuse. It supports the software reuse from two points of view: First, it has code and interface generators which can be counted as a reuse and second, it allows the software designer to follow the software life cycle, so that the developed software can be counted as reusable software because of the systematic

development. Rational Rose is a Computer Aided Software Engineering (CASE) tool developed by the Rational Corporation under the direction of Booch, Jacobson and Rumbaugh to support software development using UML.

(e) Pure::variants

Pure::variants is a set of integrated tools to support each phase of the software product-line development process and the most flexible technology available for variant management. Pure::variants has also been designed as an open framework that integrates with other tools and types of data such as configuration management systems, requirements management systems, code generators, object-oriented modelling tools, compilers, UML descriptions, documentation, source code, etc[37].

(f) Software Thesaurus(ST)

Software Thesaurus is a tool for reusing software objects. Software Thesaurus is use to develop software by reusing objects produced earlier in other software projects. This tool is defined by a new repository metamodel which supports the classification and retrieval of essential software objects defined by current object oriented methodologies. The reusable object for ST can be static component, dynamic component or graphic component. Static component can be attributes, variables and relationship between concepts. Dynamic components can be action, events and processes. Graphic component can be graphical classes or graphical items.

(g) Bayfront Computer-Aided Protocol Engineering (CAPE) Tools

Bayfront Technologies introduces a collection of software tools that provide software designers with an edge. Bayfront's Computer Aided Protocol Engineering (CAPE) real-time CASE tools automate certain percentage of communications systems implementation, maintenance and documentation. Bayfront CAPE Tools are used to design and implement any system that uses protocols or state machines including communications systems, real-time systems, client/server systems, operating systems, transaction systems and process control systems. Corporations are currently using Bayfront CAPE Tools in the design of a wide variety of complex systems that include communications switches, LAN hubs, A equipment, cellular network hubs, metropolitan area networks, ISDN equipment, voice/data switches, medical monitoring equipment, wireless data equipment and satellite VSAT systems[38].

6.2 Technologies

There is barrier named lack of technologies which is also identified as most critical barrier and also influence software reuse. So variety of component models have been identified in order to overcome this barrier**.**

6.2.1 Components Models

Component models try to solve the barrier and will motivate the reuse capability. A software component model is a definition of the semantics of components, that is, what components are meant to be, the syntax of components, that is, how they are defined, constructed, and represented, and the composition of components, that is, how they are composed or assembled. Different component models are:-

(a) COM

Component Object Model(COM) technology in the Microsoft Windows-family of Operating Systems enables software components to communicate. COM objects can be created with a variety of programming languages. Object-oriented languages, such as C++, provide programming mechanisms that simplify the implementation of COM objects. Microsoft provides COM interfaces for many Windows application programming interfaces such as Direct Show, Media Foundation, Packaging API, Windows Animation Manager, Windows Portable Devices, and Microsoft Active Directory[39]. Component Object Model is a platform independent, distributed, object-oriented software architecture for creating and connecting binary software components[40]. COM defines how components and their clients interact.. This interaction is defined such that the client and the component can connect without the need of any intermediate system component[41]. It evolved in two directions, with COM+ toward a simpler component model for in-process components and DCOM for distributed components[14]. As an extension of the Component Object Model (COM), Distributed COM (DCOM), is a protocol that enables software components to communicate directly over a network in a reliable, secure, and efficient manner.

(b) AUTOSAR

AUTomotive Open System Architecture(AUTOSAR) is an open and standardized automotive software architecture, jointly developed by automobile manufacturers, suppliers

and tool developers. The AUTOSAR standard will serve as a platform for future vehicle applications[42].The goal of AUTOSAR is to provide a way for managing increasing complexity of vehicular embedded systems, enable detection of errors in early design phases and improve flexibility, scalability, quality and reliability of such systems [43]. In AUTOSAR, application software is organized in independent units, called software components. Such components hide the implementation of the functionality and behaviour they provide and simply expose very well defined connection points, called ports. The software component of an AUTOSAR system do depend on each other with respect to utilizing each other's functionalities. However, those kind of dependencies are described precisely in form of interfaces and ports, and no internal, hidden dependencies may exist[44].

(c) CORBA Component Model(CCM)

CORBA is an open standard for application interoperability that is defined and supported by the Object Management Group (OMG). CORBA component model started out as a component model for distributed objects. The most important part of a CORBA system is the Object Request Broker (ORB) architecture. The ORB is the middleware that establishes the client-server relationships between components[45]. Components in the CCM can be implemented in any programming language and on any platform as long as they use the CORBA middleware. A component package in the CCM consists of compiled program code (e.g. library file for C++ or class file for JAVA), and a CORBA component descriptor.

(d) BIP

Behavior, Interaction, Priority(BIP) is a framework used for modelling heterogeneous real-time components and was developed at Verimag[46]. In BIP the heterogeneity can refer to either **synchronous** or **asynchronous** and **timed** or **untimed** components. BIP models are non-deterministic and fully characterize the behavior of the wireless sensor network, independent of the used platform .It supports a component construction methodology in which the components are obtained as the superposition of three layers. The lower layer describes behavior. The intermediate layer includes a set of connectors describing the interactions between transitions of the behavior. The upper layer is a set of priority rules describing scheduling policies for interactions. It allows considering the system construction process as a sequence of transformations in a three dimensional space: Behavior × Interaction × Priority[47].

(e) EJB

JavaBeans component model is a component model for customizable reusable components [14]. Sun's Java-based component model consists of two parts: the JavaBeans for client-side component development and the Enterprise Java Beans (EJB) for the server-side component development. The JavaBeans component architecture supports applications of multiple platforms, as well as reusable, client-side and server-side components[48]. It envisions the construction of object-oriented and distributed business applications. The model simplifies the development of middleware by providing server support for a set of services, such as transactions, security, persistence, concurrency and interoperability. EJB specification introduces three kinds of components called beans: Entity beans, Session beans and Message – driven beans.

(f) FRACTAL

The FRACTAL component model is a general component model developed by France Telecom R&D and INRIA that is intended to implement, deploy, and manage complex software systems, including in particular operating systems and middleware[49]. A FRACTAL component is a runtime entity that is encapsulated, has a distinct identity, and that supports one or more interfaces. Fractal can be used with any programming language and can be applied to variety of systems and applications from operating systems, middleware platforms to graphical user interfaces. The JULIA framework supports the construction of software systems with FRACTAL components written in Java. The main design goal for JULIA was to implement a framework to program FRACTAL component membranes.

(g) Koala

Koala is developed by Philips which is a specialized component model and architectural description language. It includes embedded software, more specifically consumer electronics. Philips software architects and developers use it to develop software for their mid- and high-range TV sets. Koala is intended to handle the diversity and complexity of embedded software, at an increasing production speed, by using and reusing software components within an explicit software architecture [50]. Koala supports all stages of the lifecycle.Main entities of Koala are components, interfaces, configurations, modules and switches.

(h) KobrA

KOmponenten**B**asie**R**te **A**nwendungsentwicklung(KobrA) method represents a synthesis of several advanced software engineering technologies, including product line development, component based software development, frameworks, architecture-centric inspections, quality modelling, and process modelling. These have been integrated in KobrA with the basic goal of providing a systematic approach to the development of high-quality, component-based application frameworks[51]. KobrA is a general-purpose software engineering method for the development of component-based application frameworks.

(i) PECOS

PErvasive COmponent Systems(PECOS) is a component model, which is suitable for embedded devices. This model is supported by the COCO language, which serves to specify components and system architectures[52]. The goal of PECOS is to enable CBSD for embedded systems by providing an environment that supports the specification, composition, configuration checking, and deployment of embedded systems built from software components[53]. CoCo is used in PECOS for the specification of components and for the composition of components to build entire field device applications[54].

(j) Pin

Pin is a basic, simple component technology suitable for building embedded software applications which is developed at Carnegie Mellon Software Engineering Institute. Pin is a component technology for pure assembly—systems are assembled by selecting components and connecting their interfaces (composed of communication channels called pins)[55]. Pin has since been further developed for use in prediction-enabled component technologies (PECTs). Pin software components consist of prefabricated containers that provide a standardized interface, and custom code that they implement internally. Pin systems are modelled using ADL-like construction and composition language (CCL). Their implementation is defined by the C programming language, and such software components are packaged as .DLL files.

(k) Robocop

Robocop Component-Based Architecture (CBA) is used for conducting research on predictable software design, because it offers software reuse and speeds up the development.

The Robocop architecture is developed for middle-ware in consumer devices, with the emphasis on robustness and reliability. It aims to support definition, modelling and trading of software components, their use in consumer electronics applications, and run-time upgrades and reconfiguration of such applications. A Robocop component is a set of possibly related models such as resource model, behaviour model, functional model and executable model[56].

(l) Rubus Component Model

Rubus is developed by Arcticus systems, with support from the research community which is collection of methods and tools. The Rubus component model is tailored for resource constrained systems with real-time requirements[57]. Rubus has a red and a blue part for hard and soft real-time respectively. The red kernel is used for time-critical applications and is therefore time triggered. The blue kernel is event-triggered, and used for less time-critical applications. The computation model provided by Rubus is the desired pipe and filter model, very simple and suitable for control applications. Like Koala, Rubus also has source-code components. The components are hence open for inspection and white-box testing.

(m) OpenCOM

OpenCOM is a component model developed at Lancaster University. OpenCOM v1 was originally described, but it had a number of revisions. Currently OpenCOM v2 is under development. OpenCOM v1 was based on a subset of Microsoft's COM, however, OpenCOM v2 removes its reliance from COM, as it aims to be truly platform independent. It is a general-purpose component model for construction of low-level systems software, such as embedded systems, operating systems, communications systems, programmable networking environments or middleware platforms.

(n) Palladio Component Model

Palladio Component Model (PCM), a novel software component model for business information systems, which is specifically tuned to enable model-driven quality-of-service predictions. The PCM’s goal is to assess the expected response times, throughput, and resource utilization of component-based software architectures during early development stages[58]. Two key features of the PCM are[46] the parameterised component QoS specification and the developer role concept.

(o) ProCom

The ProCom component model is specifically developed to address the particularities of the embedded systems domain, including resource limitations and requirements on safety and timeliness[59].

ProCom is an industrial component model for development of control software for embedded systems. The language consists of two layers called ProSys and ProSave for component modeling of the software. While ProSys models higher level module (system) structure, ProSave layer is used for functional modeling of each module (subsystem)[60]. The kind of complex distributed embedded systems found in the targeted domains typically have quite different characteristics when considered at different levels of granularity. The big parts of the system are different from the small parts, in terms of execution model, communication style, synchronisation, etc. ProCom is a component model for control-intensive distributed embedded systems and is designed to cover the whole development process in the vehicular-, automation- and telecommunication domains[61].

(p) SaveCCM

SaveComp Component Model(SaveCCM) [46] is a research, domain-specific component model developed at Malardalen University. It is intended for embedded control applications for vehicular systems, mainly considering the safety-critical subsystems responsible for controlling vehicle dynamics such as power-train, steering, braking, etc. It is a simple component model that limits the flexibility of modeling to enable analyzability with respect to timing. SaveCCM consists of the four main elements namely components, switches, assemblies and run-time framework.

(q) SOFA

Software Appliances – SOFA is a component model developed within the SOFA academic project at Charles University in Prague [46]. It encompasses several software domains, such as the communications middleware, component management, component design and security. Key issues addressed by SOFA include component transmission protocol, dynamic component downloading and updating, hierarchical top-down design, distributed deployment and versioning, and support for component trading and licensing.

In SOFA, a system is built out of a set of dynamically updateable components. Every SOFA component is specified by its frame and architecture. The frame provides a black box view of

a component through the provided and required interfaces. SOFA architecture is an implementation of a frame. Every frame can be implemented by more than one architecture.

(r) COMDES-II

COMponent-based design of software for Distributed Embedded Systems, version II(COMDEs-II) employs a generative programming software engineering methodology, which can be used to automate the generation of system implementations from higher-level abstractions represented as textual or graphical models [62]. COMDES-II is a component-based software framework intended for model-integrated development of embedded control systems with hard real-time constraints[63].

COMDES-II defines a two-layer component model. Components in the first layer are called actors. Actors are active software artifacts consisting of multiple I/O drivers, which define their port-based interface, and a single actor task. actuating from/to physical units. In the second layer of component model, as specification of functional behaviour of actor tasks.

COMDES-II uses function block instances, which are instantiations of function block types[46].

(s) BlueArX

BlueArX [64] is a domain-specific component model developed and used by Bosch for real-time embedded automotive applications, for example in engine control systems or chassis systems. These are closed control loop systems, meaning that they receive physical values from sensors, perform computations and then control actuators with new physical values.BlueArX provides support in all stages of the lifecycle.BlueArX supports two types of components: atomic and structural and BlueArX divides interfaces into two types: import and export.

6.2.2 Table of comparison

Generally component Model define how components are specified and connected but components models differs by their features. There are certain parameters which show comparison between different component Models. Table 6.1 shows comparison of components models

Table 6.1 : Comparison of Component Models

S.NO	Parameters		Component Models
1.	Language Independence		CCM, COM, CompoNETS, KobrA, OpenCOM
2.	Ease of packaging	Whitebox	Fractal, Koala, KobrA
		Blackbox	CCM, COM, CompoNETS, EJB, JavaBeans, PECOS, Pin
3.	Integration mode	Static binding	AUTOSAR, BIP, BlueArX, COM, COMDES-II, JavaBeans, Koala, KobrA, Pecos, Pin
		Dynamic binding	CCM, COM, CompoNETS, EJB, Fractal, OpenCOM, Palladio
4.	Communication type	Synchronous	JavaBeans, Koala, kobrA, OpenCom, Palladio, PECOS, Rubus, SaveCCM,
		Asynchronous	AUTOSAR, BIP, BlueArX, CCM, COM, CompoNETS, EJB, Fractal, Pin, ProCom, ROBOCOP, SOFA 2.0
5.	Interaction Mode	Operation based	AUTOSAR, BlueArX, CCM, CompoNETS, EJB, Fractal, JavaBeans, Koala, KobrA, OpenCCM, Palladio, SOFA 2.0
		Port based	BIP, AUTOSAR, BlueArX, COMDES-II, PECOS, Pin, ProCom, ROBOCOP, Rubus, SaveCCM
6.	Domain	Specialized	AUTOSAR, BIP, BlueArX, COMDES-II, Koala, Palladio, PECOS, ProCom, ROBOCOP, Rubus, SaveCCM
		General-Purpose	CCM, COM, CompoNETS, EJB, Fractal, JavaBeans, KobrA, OpenCOM, Pin, SOFA 2.0

The table shows that not all the component models are language independent. If the reuser of the component needs independence of language, then CCM, COM, CompoNETS, KobrA and openCOM can be used. Fractal, Koala and KobrA are easy to package when white box reuse is concerned. COM and EJB can be used for asynchronous communication, COMDES-II can be used for port based interaction. JavaBeans, EJBs, Pin and SOFA models can be used for general purpose component development while AUTOSAR, BIP, PECOS and ROBOCOP are models for developing specialized components.

6.3 Methodologies

One of the most critical barrier is lack of methodologies. In order to overcome this barrier, different programming paradigms has been defined to facilitate software reuse. First there was object-oriented programming technique to support reuse. OOP was although successful in modelling and implementing but still it has certain limitations, so aspect-oriented programming language was introduced to overcome the limitation of OOP. After that subject-oriented programming and view-oriented programming language was developed to support software reuse.

6.3.1 Object-Oriented Programming Characteristics

Object-Oriented Programming(OOP) is known to be a one of the leading programming technique we are using and can be mostly seen to be used with current software process and development tools. OOP supports software reuse by providing design and language constructs for modularity, encapsulation, inheritance, and polymorphism[65].OOP is a technique in which software system is decomposed into subsystem based on objects and computation is done by exchanging messages among objects.

OOP Limitations

OOP is famous and effective programming technique. Although, OOP was successful in modelling and implementing but still it has certain limitations. Experience with large products suggests that programmers may face some problems with maintaining their code as it is difficult to cleanly separate concerns into modules. some design decision cannot illustrated with the object oriented model[30]. Limitations are:-

OOP does not allow strong implementation reuse.

Software properties cannot be isolated in a single functional unit instead they crosscut multiple components i.e. logging, security checks, transaction management, etc. are all over the place.

Crosscutting concerns result in tangled code which is hard to develop and maintain[23].

6.3.2 Aspect Oriented Programming(AOP) characteristics

There are many programming problems where OOP techniques are not sufficient to clearly capture all the important design decisions the program must implement. A new programming technique, called aspect-oriented programming(AOP) can be used, which provides better separation of concerns and enhances the reusability[66]. AOP was introduced to solve crosscutting problems of OOP. It is originated in academic/research community in 1997 and is primarily researched and developed by Xerox PARC.

(a) AOP Terminology

Here is the different terms used the AOP[30,67].

Aspect

As classes are for OOP, in the same way aspects are for AOP. It can extend other aspects. The Aspect associates join points/point-cuts/advice and applies introductions.

Advice

Advices are the executable part of the aspects. Advice is the code that implements a concern and is the additional code we want to apply. It defines what code to run when the join point is fired. There are three main kinds of advices: before, after and around.

Join point

It is with join points you decide where the aspects is executed. AspectJ includes 11 different join points. It is a location in code where advice can be executed.

Point-cut

It identifies sets of join points and is the point of execution in the application at which cross cutting concerns need to be applied.

Introduction

Introduction also known as inter-type declaration. Introduction modify a class to add fields, methods or constructors and also modify a class to extend another class or implement a new interface.

Concern

Concern is a particular goal or area of interest. Concern define the functionality to be consolidated. Examples of concern are authentication, caching, context passing, error handling, debugging, synchronization, transactions.

Crosscutting

Crosscutting refers to what aspects do to application classes.

Weaving

The process of inserting aspect code into other code. The weaving process can done at compile-time, load time and run time. A separate aspect compiler is used in compile-time weaving. In load-time weaving the class loader is responsible for the weaving process while loading the classes into the virtual machine. The rum-time weaving uses proxy classes and code generation libraries. Compile-time aspect language implementation is the AspectJ, while the Spring Framework offers the most well known runtime implementation[68].

Instrumentor

Instrumentor is basically a tool which performs weaving in the efficient way.

(b) Working of AOP

Aspect oriented Programming consist of three development stages: Aspectual decomposition, Concern identification and Aspectual recompostiton. The working is shown in figure 6.1. First we identify cross cutting concerns and common concerns by decomposing the requirements. Then each concern is implemented separately[30]. At last there comes weaving, process of causing a relevant advice at each join point to be executed[68]. The recomposition process also known as weaving or integrating.

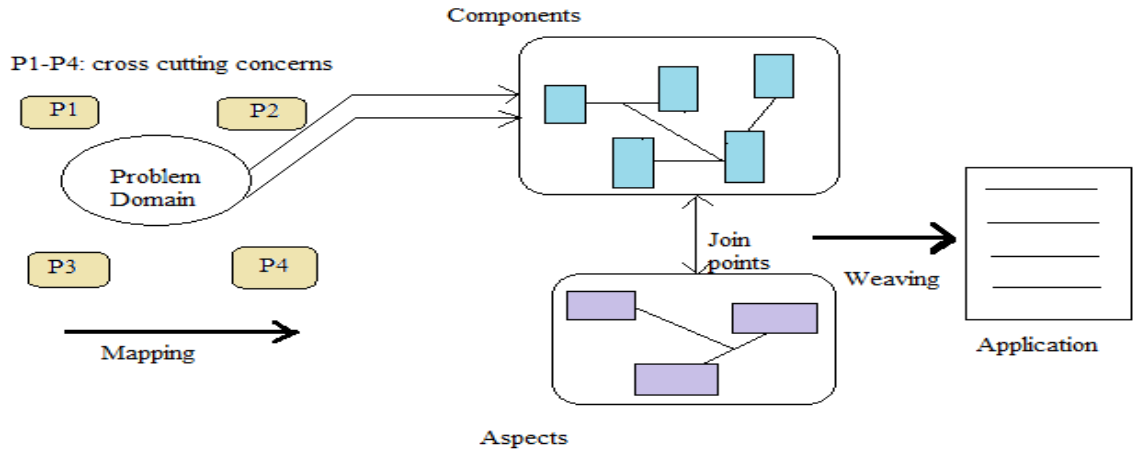


Figure 6.1: Working of AOP

(c) Aspect Weaver

A program developed using AOP has little bit different compilation process than general compiling. Before the program is compiled into an executable, AOP lets code and aspects to

be woven together by an aspect weaver. Aspect weaver work by generating a join point representation of the component program, and then executing the aspect programs with respect to it. The weaving process can done at compile-time, load time and run time[30].

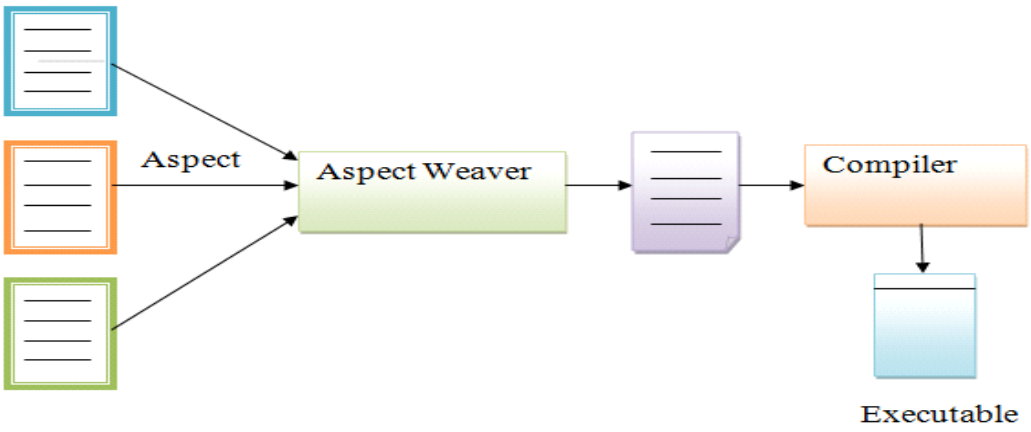


Figure 6.2 : Weaving process

(d) Tools for AOP

AOP has been implemented in different languages like C++, smaltalk, C#, C, java. There is list of tools that support AOP with java like AspectJ, JAC, ArchJava, JMangler, Hyper/J, AspectWerkz, MixJuice, PROSE etc[65].

(e) AspectJ

AspectJ is an extension of Java language for AOP and is created at Xerox Parc. AspectJ includes join points, aspect, advice, pointcut. Figure 6.3 shows the summary of AspectJ construct[69].

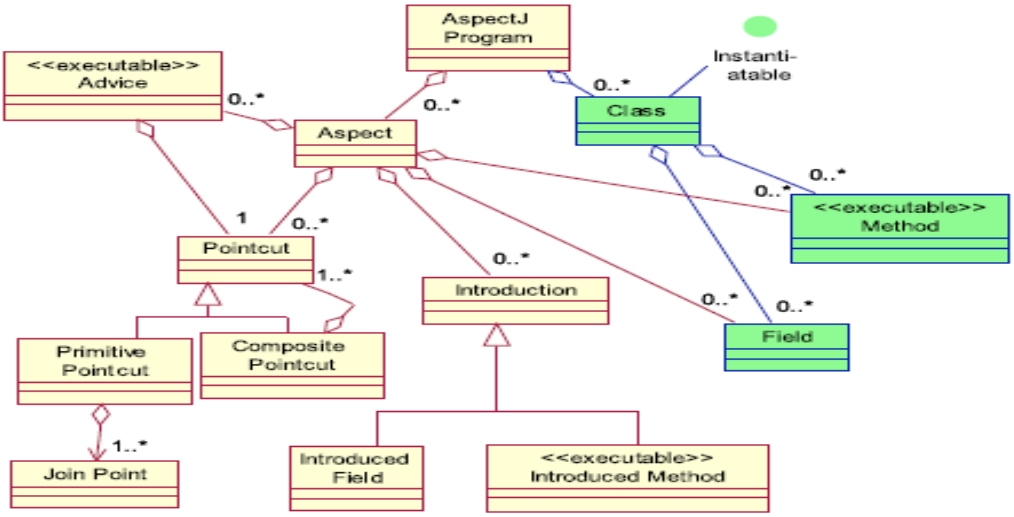


Figure 6.3: AspectJ construct Summary

Java implementation of AOP is AspectJ. Here is small example to show strength in AspectJ
First, we have a simple class containing two methods for printing messages:

```
// Module.java
public class Module
{
      public static void Modulelanguage( )
       {
           System.out.println(“Aspect-Oriented Programming”);
       }
      public static void Selectlanguage(String language)
       {
            System.out.println(“Software reuse” +language);
       }
}
```

Before the program prints “Aspect-Oriented Programming”, it should print “Object-oriented programming”, and after the message, it should print “Subject-Oriented Programming”. So, this is the implementation :

```
// LanguageAspect.java
public aspect LanguageAspect
{
     pointcut callLanguage( ) : call(public static void Module.module*(..));
     before( ) : callLanguage( )
     {
          System.out.println(“Object-Oriented Programming”);
      }
      after( ) : callLanguage( )
      {
          System.out.println(“ Subject-Oriented Programming”);
      }
}
```

In this example, there are two method calls : Modulelanguage and Selectlanguage and the two advices are being used before and after reaching the callLanguage which would print “Object-oriented programming” and “Subject-Oriented Programming”.

(f) Benefits of AOP

- AOP addresses each concern separately with minimal coupling, resulting in modularized implementations of crosscutting concerns[23,30].
- AOP overcomes the problem caused by code tangling and code scattering[30].
- Architects can delay making design decisions for future requirements since they can be implemented as aspects[23].
- AOP implements each aspect as a separate module, each individual module is more loosely coupled and results in more code reusability and also reduce the redundancy of code[23,30]. A case study is shown in appendix B.
- Aspected modules can be unaware of crosscutting concerns, so it is easy to add newer functionality by creating new aspects[23].
- AOP used for developing simple and cleaner code and also easy to maintain[65].

(g) Limitations of AOP

- As AOP is a new technology, so it is not very well tested and documented.
- It is difficult to understand a software because of invisibly injected aspects.
- It has fragile build problems.
- It has a complicated control flow breakage.
- There may be a chance for reduce quality of software if aspects are not appropriately managed.
- It can increase the complexity of the system as one must understand the functionality of both base and the aspect programs in order to understand the whole system.
- Faulty pointcut definitions may cause advices to be bound to wrong join points or not to any join points at all.

6.3.3 Subject-Oriented Programming characteristics

Subject-oriented programming(SOP) is a practical approach to object-oriented programming in large. It addresses some of the limitations of OO development. These limitations include weaknesses in:

Non-invasive system extension and evolution: Creating extension to software and configurations to software without modifying original source code and keeping details for multiple platforms, versions and features separate.

Large-Scale reuse and integration: OO development has insufficient mechanism to achieve large-scale reuse or integration of off-the shell components without significant preplanning.

Multi-team/ decentralized development:- OO development leads to contention over shared, centralized classes. It forces developers to agree on single domain model, rather than using models appropriate to their tasks.

Standard OO techniques, like subclassing, frameworks and design patterns improve some of the problem but do not solve them properly. A major limitation is that the kind of flexibility they provide require preplanning. SOP allow OO systems to be built by flexible composition of components which we call "subjects"[69]. A subject is a code packaging consisting of collection of class definitions. These define the subject's particular view of domain. Some subjects are complete and executable programs. Others are incomplete, consisting of fragments of classes implementing some subset of the total functionality. Incomplete subject must be composed with other subjects before they can be executed. Subject oriented project contain many subjects that will be composed together[70].

(a) Limitations of Subject-oriented programming

- The entry-level subject composition is too hard. Unlike aspect-oriented programming, which provides very simple features that perform a very useful service, subject-oriented programming lacks this gradualness.
- The combination of inheritance hierarchies is a difficult problem, regardless of the language.
- Methods may be too coarse-grained to yield effective behavioural compositions[14].

6.3.4 View-Oriented Programming characteristics

View-Oriented Programming is our own home grown approach for developing applications by composing independently developed functional slices. There is interesting difference between view programming and the previous approaches. First, an essential requirement of or approach is the ability to add and remove view(aspects, subjects) dynamically to an object, that is, during runtime. This is not possible in either aspect-oriented programming or subject-oriented programming, where composition takes place at the source code level before programs are compiled, linked and run. View oriented programming supports[71]:-

- Allow client program to access several functional areas or views simultaneously
- Addition and removal of views during run-time.

- Consistent protocol to address objects that support views

In our approach, an application object is considered as an aggregation of components consisting of core object that embodies the application object's basic data and behaviour (and unique identity), and a time varying set of views, which are object fragments that add usage-specific or role-specific data and functionality. These fragments(views) rely on some of the basic services of the core object. The response of an application object to a message depends on the set of views that are currently attached to it; for a given message and any given point in time[14].

CHAPTER 6

CONCLUSION & FUTURE SCOPE

The aim of the thesis is to improve software reuse by identifying the barriers related to software reuse and also provide the solution for those barriers. Many companies have setup successful reuse programs but not all the reuse programs are successful due to these barriers. So critical barriers are identified along with the reasons behind these and a framework is provided for handling these barriers.

7.1 Conclusion

In order to decrease the time and effort of the software development process and to increase the quality of the software, software engineering required technologies. So software engineering design is based on reuse of existing system or components. Despite the benefits of software reuse, it is not widely used by the organization. There are many factors that directly or indirectly influence the success of reuse. These factors can be technical, managerial or cultural. There are many barriers which impede the implementation of software reuse. So different barriers to software reuse and reason behind them are identifed. The suggested framework defined provides the solution to overcome those barriers.

7.2 Future Scope

Different barriers have been identified with the reason behind them and also different ways to overcome those barriers. So software reuse can be improved by applying the identified solutions to the barriers.

REFERENCES

[1] Eunjung Lee, "Software reuse and its impact on Productivity, Quality and Time-to-market", Department of Computer Science University of Houston, www2.cs.uh.edu/~ejlee/ejlee.pdf.

[2] Michael L.Nelson, "Barriers to Software Reuse and the Projected Impact of World Wide Web on Software Reuse", 1996, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=?doi=10.1.1.43.7165.

[3] B. Jalender, A. Govardhan, P. Premchand, "A Pragmatic Approach To Software Reuse", Journal of Theoretical and Applied Information Technology (JATIT), Vol. 14, No. 2, pp.87-96,June2010.

[4] Peter Freeman, "Reusable software engineering concepts and research directions" In Tutorial: Software Reusability, pp. 10-23, 1983.

[5] Victor R. Basili and Hans Dieter Rombach, "Towards a comprehensive framework for reuse: A reuse-enabling software evolution environment", Technical Report CS-TR-2158, University of Maryland, December 1988.

[6] Christine L. Braun, "Reuse", Encyclopedia of Software Engineering, pp. 1055-1069, 1994.

[7] Will Tracz, "Confessions of a Used Program Salesman: Instituionalizing Software Reuse" Addison-Wesley, 1995.

[8] Jack Cooper, "Reuse-the business implications" In Encyclopedia of Software Engineering", pp. 1071-1077, 1994.

[9] Wayne C. Lim, "Effects of reuse on quality, productivity, and economics" IEEE Software, 11(5):23(8), September 1994.

[10] Sametinger, "Software Engineering with Reusable Components", Springer-Verlag, ISBN 3-540-62695-6, 1997.

[11] M. R. J. Qureshi, S. A. Hussain, "A reusable software component-based development process model", Advances in Engineering Software, Vol.39, No. 2, pp.88-94, February 2008.

[12] Kyo C. Kang, "Issues in Component-Based Software Engineering", Asia-Pacific Software Engineering Conference(APSEC) 1999 , pp. 394-403.

[13] Iqbaldeep Kaur, Parvinder S. Sandhu, Hardeep Singh, and Vandana Saini "Analytical Study of Component Based Software Engineering" pp.437-442, 2009.

[14] Hafedh mili, Ali mili, Sherif Yacoub and Edward Addy, “Reuse based Software engineering”, ISBN 0-471-39819-5, 2002.

[15] Anas Bassam AL-Badareen, Mohd Hasan Selamat, Marzanah A. Jabar, Jamilah Din, Sherzod Turaev, “Reusable Software Component Life Cycle”, International Journal of Computers, Issue 2, Vol 5, pp. 191-199, 2011.

[16] B.Jalender, N.Gowtham, K.Praveenkumar, K.Murahari, K.Sampath ,“Technical Impediments to Software Reuse” International Journal of Engineering Science andTechnology (IJEST) , Vol. 2, No. 11, pp. 6136-6139, Nov 2010.

[17] William B. Frakes and Michael Tortorella, “ Foundational Issues in Software Reuse and Reliability”, http://ie.rutgers.edu/resource/research_paper/paper_04-002.pdf.

[18] M. Madiajagan, and B. Vijayakumar, “Interoperability in Component Based Software Development” World Academy of Science, Engineering and Technology , Vol. 22 pp.68-75, 2006.

[19] “Breaking Down the Barriers to Software Component Technology” by Chris Lamela IntellectMarket, Inc., http://www.docstoc.com/docs/44301665/Breaking-Down-the-Barriers-to-Software-Component-Technology.

[20] G.Sindre, R.Conradi, Even Karlsson, “The Reboot Approach to Software Reuse”, Journal of Systems and Software, Vol. 30 , pp. 201–212, 1995.

[21] G. Goh Guan, W. Kim Yen, M. Toleman, “Software component reuse in information systems development: a review of challenges and strategies”, Journal of Han Chiang College, Vol.3, pp. 83-95, 2005.

[22] G. T. Heineman, W. T. Council, “Component-Based Software Engineering: Putting the Pieces Together”, Addision- Wesley Professional. 1st ed., May 2001.

[23] James Holmes “Aspect Oriented Programming in Java”, Atlanta Java Users Group July 15, 2003.

[24] Anurag Mendhekar, Gregor Kiczales, John Lamping, “RG: A Case-Study for Aspect-Oriented Programming” Technical report SPL97-009 P9710044 Xerox Palo Alto Research Center. February 1997.

[25] Brian W. Holmgren, “Software reusability: A study of why software reuse has not developed into a viable practice in the Department of Defense,” Masters Thesis, Air Force Institute of Technology, AFIT/GSM/LSY/90S-16, September 1990.

[26] M. Pat Schuler, “Increasing productivity through Total Reuse Management (TRM),” Proceedings of Technology 2001: The Second National Technology Transfer Conference and Exposition, Volume 2 , Washington DC, pp. 294-300, December 1991.

[27] Pamela Samuelson, “Is copyright law steering the right course?,” IEEE Software, pp.78-86, September 1988.

[28] Michael L. Nelson, et al., “The NASA Technical Report Server”, Internet Research: Electronic Networking Applications and Policy, Vol 5 , Number 2, pp. 25–36, 1995.

[29] Scott Henninger, “Information Access Tools for Software Reuse”, Journal of Systems and Software, Second Quarter, 1995.

[30] Niklas Pahlsson, “Aspect-Oriented Programming : An Introduction to Aspect-Oriented Programming and AspectJ” , Topic Report for Software Engineering, 2002.

[31] William Frakes, Carol Terry, “Software Reuse: Metrics and Models” , ACM Computing Surveys, Vol. 28, No. 2, June 1996.

[32] Muthu Ramachandran, “Automated Improvement for Component Reuse”, Leeds Metropolitan University LEEDS, UK.

[33] Muthupandi Ramachandran, “An Investigation into Tool Support for the Development of Reusable Software”, A thesis submitted for the degree of Ph.D at Department of Computing, The University of Lancaster, United Kingdom, March 1992.

[34] “Interaction between ARCSeeker and Enterprise Architect”, by SparxSystems, Japan.

[35] http:// www.arcseeker.com

[36] Dennis Smith, “Migration to Service Oriented Architecture (SOA) with Selected Research Challenges”, Software Engineering Institute, Carnegie Mellon University, December 2008.

[37] pure::variants User's Guide : Version 3.0 for pure::variants 3.0, pure-systems GmbH.

[38] http://www.bayfronttechnologies.com/capeprod.htm

[39] Microsoft, COM: Component Object Model Technologies, http://www.microsoft.com/com/default.mspx

[40] Wang Yongjun, Gong Jianya, “ A COM based framework for management and visualization of large-scale dems”, in proceedings ARCS 2002.

[41] X. Cai,, M. R. Lyu and K-F. Wong Roy-Ko,"Component-Based Software Engineering: Technologies, Development Frameworks, and Quality Assurance Schemes", International Journal of Software Engineering and Knowledge Engineering, 2000.

[42] AUTOSAR: http://www.autosar.org

[43] AUOTSAR Development Authority, AUTOSAR – Technical Overview, V2.2.1,2008

[44] AUTOSAR Development partnership, AUTOSAR Software Component Template V2.0.1, 2006

[45] OMG:http://www.omg.org/corba/whatiscorba.html, Mar, 2000.

[46] Juraj Feljan, Luka Lednicki, Josip Maras, Ana Petričić, Ivica Crnković "Classification and survey of component models" , DICES technical report Contract No. 03/07, Feb 2009.

[47] A. Basu, M. Bozga, J.Sifakis, "Modeling Heterogeneous Real-time Components in BIP",

[48] SUN:http://developer.java.sun.com/developer, March 2000.

[49] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Qu´ema and Jean-Bernard Stefani "The FRACTAL component model and its support in Java", pp. 1257-1284, Software Practices Exper. 2006.

[50] Rob van Ommering, Frank van der Linden, Jeff Kramer, Jeff Magee, "The Koala Component Model for Consumer Electronics Software", IEEE Computer, 2000.

[51] C. Atkinson, J. Bayer, D. Muthig "Component-Based Product Line Development: The KobrA Approach", 1st International Software Product Line Conference, Pittsburgh, Pennsylvania, 08. 2000.

[52] Michael Winter, Christian Zeidler, Christian Stich, "The PECOS Software Process", IST Program IST-1999-20398.

[53] Oscar Nierstrasz, Gabriela Arevalo, Stephane Ducasse, Roel Wuyts, Andrew P. Black, Peter O. Muller, Christian Zeidler, Thomas Genssler, Reinier van den Born, "A Component Model for Field Devices", Lecture Notes in Computer Science, 2002.

[54] Michael Winter, Thomas Gensler, Alexander Christoph, Oscar Nierstrasz, Stephane Ducasse, Roel Wuyts, Gabriela Arevalo, Peter Muller, Chris Stich, Bastiaan Schonhage, "Components for Embedded Software - The PECOS Approach", The Second International Workshop on Composition Languages, in conjunction with the 16th ECOOP, 2002.

[55] Scott Hissam, James Ivers, Daniel Plakosh, Kurt C. Wallnau, "Pin Component Technology And Its C Interface", Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, V1.0, 2005.

[56] Egor Bondarev, Peter de With, Michel Chaudron , Johan Muskens “Modelling of Input-Parameter Dependency for Performance Predictions of Component-Based Embedded Systems”, In Proceedings of 31th Euromicro Conference, Component-based Software Engineering Track, Porto, Portugal; September, 2005

[57] Anders Möller, Mikael Åkerholm, Johan Fredriksson, Mikael Nolin “Software Component Technologies for Real-Time Systems”, in TCCS Proceedings of ICALP’92, Lecture Notes in Computer Science.

[58] Ralf Reussner, Steffen Becker, Jens Happe, Heiko Koziolek, Klaus Krogmann, Michael Kuperberg, “The Palladio Component Model”, Technical report, University of Karlsruhe, 2007.

[59] Aneta Vulgarakis, Jagadish Suryadevara, Jan Carlson, Cristina Seceleanu and Paul Pettersson, “Formal Semantics of the ProCom Real-Time Component Model”, 35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2009), IEEE, Patras, Greece, August, 2009.

[60] Anuradha Suryadevara, “Towards Transforming ProCom Component Model into UML Activity Diagrams” Anuradha Suryadevara Mälardalen University Västerås, Sweden http://www.idt.mdh.se/kurser/ct3340/ht09/ADMINISTRATION/IRCSE09-submissions/ircse09_submission_20.pdf

[61] T. Bureš, J. Carlson, S. Sentilles, A. Vulgarakis, “ProCom - ProCom | the Progress Component Model Reference Manual”, V1.1, June 2010.

[62] Ke Xu, Sierszecki Krzysztof, Angelov Christo, “COMDES-II: A Component-Based Framework for Generative Development of Distributed Real-Time Control Systems”, RTCSA '07: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2007.

[63] Ke Xu, Pettersson Paul, Sierszecki Krzysztof, Angelov Christo, “Verification of COMDES-II Systems Using UPPAAL with Model Transformation”, RTCSA '08: Proceedings of the 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2008.

[64] Ji Eun Kim, Rahul Kapoor, Martin Herrmann, Jochen Haerdtlein, Franz Grzeschniok, Peter Lutz, “Software Behavior Description of Real-Time Embedded Systems in Component Based Software Development”, ISORC '08: Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing, 2008.

[65] Rani Augustine, “Report on Aspect-oriented Programming” , Department of Computer Science, Cochin University of Science and Technology, 2007.

[66] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.Videira Lopes, J. Loingtier, J. Irwin, “Aspect-Oriented Programming”, in proceedings of the European Conference on Object-Oriented Programming (ECOOP) Finland. Springer-Verlag LNCS 1241, June 1997.

[67] Mark Volkman, “Aspect-Oriented Programming in Java”, Partner Object Computing, Inc. (OCI), August 14, 2003.

[68] Jyri Laukkanen, “Aspect-Oriented programming”, University of Helsinki ,Department of Computer Science, February 2008.

[69] James Heliotis, “Aspect-OrientedProgrammingin AspectJ”, http://ebookbrowse.com/03-aspectj-pdf-d341153508

[70] Harold Ossher, Peri Tarr, “Using Subject-Oriented Programming to Overcome Common Problems in Object-Oriented Software Development/Evolution”, pp. 687-688, 1999.

[71] Support for Subject-Oriented Programming in C++ on IBM Visual Age for C++ , Version 4.

[72] Hafedh Mili, Hamid Mcheick, Salah Sadou, “Distribution and aspects” , Workshop on Aspect-Oriented Software Development, Bonn, Germany, February 2002.

PAPER PUBLISHED/COMMUNICATED

1. Jyotsna, Shivani Goel, "Critical Barriers To Software Reuse", IEEE Conference on Cognizance of Applied Engineering and Research(IEEE-ICAER'11), UIET Chandigarh, October 2011.(Published).

2. Jyotsna, Shivani Goel, "Aspect Oriented Programming: An edge over Object Oriented Programming" International Conference on Emerging Trends in Engineering and Management, Satpriya Group of Institutions, Rohtak (ICETEM 2012), May 2012(Accepted).

3. Jyotsna, Shivani Goel, "Component Bases Development Technologies: A Comparative View" International Conference on Emerging Trends in Engineering and Management, Satpriya Group of Institutions, Rohtak, (ICETEM 2012), May 2012 (Accepted).

4. Jyotsna, Shivani Goel, "Software Component Models: A Comparative View", Journal of Object Technology(JOT), June 2012(Communicated).

APPENDIX A

QUESTIONNAIRE

Dear User, The following Survey is done to find the critical barriers in Software reuse.

The barriers are divided into three broad categories: technical, managerial, cultural.

Please rate the following barriers in various categories in the importance of their place is software reuse. Please write the number as: 1-If it is of low importance, 2-If it is of medium importance, 3-If it is highly important.

S.No	**Technical Barriers**	**1- Low**	**2- Med**	**3-High**
1	Structure Mismatch			
2	Steep learning curve			
3	Poor cataloguing, Distribution & access methods			
4	Modification			
5	Integration			
6	Garbage reuse			
7	Lack of tools and technologies			
8	Lack of methodologies			
9	Object-Oriented Technology			
9.1	Doesn't allow strong implementation reuse			
9.2	Tangling of code			
9.3	Low level Abstraction			
10	Lack of explicit procedures			
	Managerial Barriers	**1- Low**	**2- Med**	**3-High**
1	Lack of top management support			
2	Lack of management incentives			
3	Infrastructure clash			
4	Turf Battles			
5	Never enough time & money to do right			
6	Quality issue			
6.1	Reliability			
6.2	Security			
7	Legal issue			
7.1	Data Rights			
7.2	Copyright			
8	Inadequate Organizational structure			
9	Cost of reusing & making reusable			
10	Difficult in reuse measurement			
	Cultural/Psychological Barriers	**1- Low**	**2- Med**	**3-High**
1	Apathy			
2	Not invented here syndrome			
3	Fear of the unknown			
4	Ivory Towerism			
5	Unfamiliarity with reuse tools and techniques			

APPENDIX B

CASE STUDY: REVERSE GRAPHICS

Reverse Graphics(RG) is an image processing system that applies certain image processing operations. First the RG was implemented using OOP, which was very easy to do and manageable, but the performance was low. So after that the Aspect-oriented programming was used, which reduces the limitations of OOP. AOP handles the complexities arising from such cross-cutting issues.

RG System

RG is a image processing system developed at Xerox PARC. Programs in RG are written in terms of operations on entire images, rather than working a pixel at a time. The system is a collection of primitive image filters that take one or more input images to produce an output image. The primitive filters fall into a few different categories based on the order in which the pixels in the argument images to the filters are processed. We will discuss two of them:-

Pointwise:- Pointwise operation is generally a pointwise addition and is done by combining several images by performing the same operations on each pixel operation. The pointwise operation of two images is given by:-

(pointwise ((a AA) (b BB)) (+ a b))

In this a, b are scalar variable and bounded to the pixel from AA and BB correspondingly.

Translate:- Translate operation takes the input image and shifts the pixel in the output image by a specified number of pixels in a given direction.

(translate : x 2 ((a AA)) a)

This will shift an image to the right direction and horizontally by two pixels.

Performance problem of RG System

There comes three serious performance problem while implementing RG system.

1. Redundancy in computation:- Applications make many redundant calls. This occurs when a higher level filter requests a lower level filter that has already been computed, at the request of a different higher level filter.

2. Excess memory turnover:- Each operation returns a fresh image, images were allocated at a very high rate. This implementation used automatic memory management, and since the images were large, the result was frequent, expensive garbage collection.

3. Inefficient data cache usage:- As all the intermediate results were also large images, they quickly filled up the on-chip data cache without ever actually being reused. The result wasthat accesses to the intermediate results tended to result in too many cache misses and correspondingly slow access times.

Strategies to overcome Performance problem

The first problem redundant computation can be solved with memoization. In this,each primitive filter can keep a record of input image and its corresponding output image after filtering them. Each invocation of the filter results in a comparison with inputs it has already seen and simply returns the previously computed output if there is a match.

The second and third problem , excess memory turnover and inefficient data cache usage are both due to intermediate images and this intermediate images can be eliminated by fusing loop i.e applying several primitive filters in a single iteration. Finally, the overall memory turnover can be further reduced if allocated memory is reused instead of being deallocated and then reallocated.

Object-Oriented Implementation

The natural implementation of the RG System is an Object-oriented language. The major part of the storage of an object is an array of pixel values representing the image data. One of the main reasons for using object-orientation in a system such as RG is that it becomes possible to use alternative image representations without changing the application code. Object orientation works well in managing the complexity of applications. But the problem with Object-oriented implementation is:-

- Memoization requires modifying each primitive filter and the record of the input and output images from each filter breaks automatic memory management.
- Custom memory management is not feasible, since the information needed to determine if an image is still useful is scattered throughout the program.
- The only way to express these performance improvements in the object oriented paradigm is to abandon the modularity of the original program, to write one large method that incorporates all the optimizations. This results in the kind of tangled code which is harder to read, debug, and modify than the original modular code.

Aspect-Oriented implementation

Performance problem of RG system cannot be solved properly by object oriented programming. So there comes the Aspect Oriented Programming(AOP) in order to address

the problem of RG System. A key step in an AOP solution is to identify the non-component issues and to understand what view of the computation will enable code addressing those issues to be expressed. For RG, the non-component code must be able to express the implementation techniques discussed above: memoization of computations, loop fusion, and memory management. We first look in more detail at what it takes to express each of those techniques.

The implementation of the memoization technique is, roughly, "For every message send invoking a primitive filter, note which filter is being invoked and note the identity of the inputs. If this combination is the same as for some previously seen invocation, use the result of that invocation, rather than recomputing it."

The implementation of the loop fusion technique is, roughly, "For every message send invoking a primitive filter, before computing its arguments, examine each argument and determine whether the loop structure needed to calculate the filter is compatible with the loop structure needed to calculate the argument. In that case, generate a single loop structure that computes both the argument value and the filter value, and replace the original message send with a send to the fused loop."

The memory management technique is, roughly, "Maintain a pool of free arrays. For every message send invoking a primitive filter, allocate the array to hold the output value from the free pool. Once the message send returns, note the identity of each argument array. If that array is not used in any subsequent message send, then place it in the free pool."

Conclusion

AOP not only helps in reducing the code complexity of system but it also maintains performance requirements. While OOP was not able to fulfil the performance requirement of RG System