SIMPLIFIED COMPONENT GENERATION AND RETRIEVAL IN AGILE ENVIRONMENT USING CASE TOOLS

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

Master of Engineering

in

Software Engineering

Submitted By:
JASPREET SINGH
(801031013)

Under the supervision of:
MS. ASHIMA SINGH
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, "**Simplified Component Generation and Retrieval in Agile Environment using Case Tools**", 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. Ashima Singh 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.

Jaspreet Singh

(Jaspreet Singh)

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

Ashima Singh

(Ms. Ashima Singh)

Computer Science and Engineering Department

Thapar University

Patiala

Countersigned by:

(Dr. Maninder Singh)

Head 27/6

Computer Science and Engineering Department

Thapar University

Patiala

(Dr. S. K. Mohapatra)

Dean (Academic Affairs)

Thapar University

Patiala

Thesis Acknowledgement

First and foremost, I would like to thank God. You have given me the power to believe in myself and pursue my dreams. I could never have done this without the faith I have in you, the Almighty.

I would like to express my sincere gratitude to my guide Ms. Ashima Singh for the continuous support of my M.E. thesis, for her patience, motivation, enthusiasm, and immense knowledge. Her guidance helped me in all the time of research and writing of this thesis. Besides my guide, I would like to thank Dr. Maninder Singh (Head, CSED), Dr. Seema Bawa (Dean, Student Affairs) and Dr. Parteek Bhatia for their encouragement, insightful comments, and hard questions.

I thank my fellow batch mates in Thapar University: Sandeep Kumar, Parth Patel, Kapil, Puneet Malhotra, Sukhpal Gill, Ashish Narang, Harsh Taneja, Amit Gupta, Abhishek and Ravinder for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we had in the last two years. Also I thank my friends: Kanwalpreet, Ashish Kaushal and Pardeep Balli working in IT companies, for their help and support in my thesis work.

Last but not the least, I would like to thank my family: my parents for supporting me spiritually throughout my life.

Jaspreet Singh

Jaspreet Singh

Abstract

The past decade observed a tremendous growth in the software engineering. Several new process models proposed, developed time to time and consistent work in the field of software process engineering is going on. Methodologies of software development like Reuse Based Development, Component Based Development and Agile Software Development (ASD) *etc*. are proving fertile methodologies in the software industry. These models address different related domains but they are not able to satisfy heterogeneous requirement of software industry. Due to the shift in the customer's preferences, fast delivery of software has become most prominent factor behind the success of the product and software development organizations. ASD not only shrinks the timeline in software development but cuts the cost also. ASD promises to deliver the product according to the market and customer need.

The problem arises when software reusability comes in picture. Some legacy software systems contains crucial functionalities that can be reused again and again but due to lack of documentation and design in their development, makes it difficult to extract the reusable functionality and also make it very costly and difficult to modify them for the organizations. ASD tends to be simple and domain specific accompanied with minimal support documentation. ASD generates specialized product thereby reduces the scope of reusability. Reuse relies on support documentation and favors more generalized components. Therefore, the problem of reusability arises and the development effort is wasted in agile software development. In order to incorporate reusability in Agile Environment, Reengineering based Component Generation and Retrieval model has been proposed in this thesis. The model has been divided into two phases namely ES phase (Extract and Store phase) and SREM phase (Search, Retrieve, Evaluate and Modify phase). In ES phase, the source code is supplied to a reverse engineering tool to create its design in the form of UML diagrams and using Object Oriented Analysis and Designing (OOAD) the components are identified and stored in component repository. In SREM phase, desired components are searched, retrieved, evaluated and modified according to the requirements and reused in the future projects.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Agile Software Development

This chapter talks about the Agile Software Development. Differing from other software engineering projects, Agile Software Development is a group of software development methods based on iterative and incremental development, where requirements and solutions evolve through collaboration between self-organizing, cross-functional teams. It promotes adaptive planning, evolutionary development and delivery, a time-boxed iterative approach, and encourages rapid and flexible response to change. It is a conceptual framework that promotes foreseen interactions throughout the development cycle. A software product is usually invisible [1]; Thus, software development projects are highly unpredictable and that is why so many software development projects fail to meet expectations [2]. The reason for why the Agile Software Development is needed for software development can be explained as below:

"If a predictive method like Waterfall is to avoid pain of problems, then we can say that Agile was developed to solve the pain of problems."

M. Scott Peck, The Road Less Traveled.

1.1.1 History

The Agile Software Development is a group of software process methodologies. The Agile Software Development is also known as Agile Development. It emerged in mid 1990s to react against "Heavyweight" methods such as the Waterfall Process Model [3]. Initially, the Agile Software Development was called "Lightweight Methodology". The Agile methods grew up separately for a number of years, until 2001. In February, 2001, a group of their leading proponents met at Snowbird, Utah. They agreed to the name "Agile Methodologies" and created the Agile Manifesto 2 and the principle. Later on, some members of this group formed The Agile Alliance, a non profit organization that promotes the Agile Development.

(a) The Agile Manifesto

The manifesto states that we are uncovering better ways of development software by doing it and helping others to do it. Through this work we have come to value:

i. **Individuals and interactions** over processes and tools.
ii. **Working software** over comprehensive documentation.
iii. **Customer collaboration** over contract negotiation.
iv. **Responding to change** over following a plan.

(b) The Agile Principle

List of Agile Principle is given below

i. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
ii. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
iii. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
iv. Business people and developers must work together daily throughout the project.
v. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
vi. The most efficient and effective method of conveying information to and within a development team is face to-face conversation.
vii. Working software is the primary measure of progress.
viii. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
ix. Continuous attention to technical excellence and good design enhances agility.
x. Simplicity–the art of maximizing the amount of work not done–is essential.
xi. The best architectures, requirements, and designs emerge from self-organizing teams.
xii. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

1.1.2 Characteristics of the Agile Software Development

Generally, agility is defined by ability which is able to be flexible and adaptable to change [4]. The idea of the Agile Development Framework is to create a pain free working environment for those small, co-located, self-organized teams in order to assist companies to take full advantage of the customer value of the delivered software product [5]. On an Agile project, developers work closely with their customers to understand their needs, they are placed in a pair to implement and test their solution, and the solution is shown to the customers for quick feedback [6]. Therefore, the business contract will not become a barrier between customers and developers, but a platform to help customers and developers work together (refer figure 1.1).

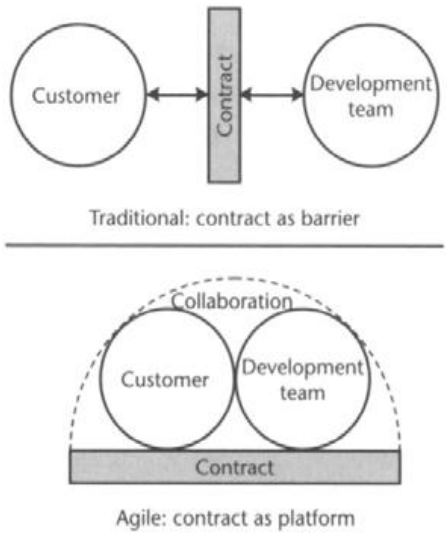

Figure 1.1: Comparison of contracts between traditional and the Agile Development Framework [8].

Moreover, the book, Managing Agile Projects, defines that the Agile Software Development is the work of energizing, empowering, and enabling project teams to rapidly and reliably deliver business value by engaging customers and continuously learning and adapting to their changing needs and environments [7]. In the Agile Development Framework, user requirements (as knows as Users Story) are written from users' perspective, elaborating on what users want to do with a feature of the software. The composition of User Stories is usually considered by business value, the story point, and other factors such as risk. Thus, using User Stories is a simple and brief way to express those user requirements. A User Story is usually composed in natural language. The construct of a User Story is as follows: As a [user

role], I want to [goal], so I can [reason]. A User Story should focus on Who, What and Why of a feature, but not How.

a) The concept of the Agile Software Development

i. **Eliminate Waste**: The Agile Development Framework advocates a "barely sufficient" approach to plan, process, and control software development process [9]. "Barely sufficient", in other words, is to find the simplest things to meet needs of requestor instead of wasting unnecessary resources.

ii. **Sustainable Pace:** The Agile Development Framework requires a daily meeting. All team members have to report what they have accomplished and what they are going to do in the meeting so that the progress of the project can be traced.

iii. **Intense Collaboration**: Unlike the traditional approach, which relies on documents, the Agile Development Framework requires a daily face to face communication with customers and co-workers to understand and fulfill their requirements.

iv. **Frequent Delivery**: Frequent delivery offers incremental business value to customers. Customers experience the growth of the system and obtain additional insight to how requirements are planned by interacting with the system early. Thus, frequent delivery is one important way to seek feedback on the quality of the application.

v. **Continuous Feedback**: The Agile Development Framework was invented to achieve customer's value. Thus, continuous feedback is demanded in order to inspect if the development team is well aligned with business objectives.

vi. **Include Change**: Differing from the traditional approach, change of requirements is considered in the Agile Development Framework since remaining adaptable is a key to building a trusting relationship with customers. Furthermore, prioritizing features, exploring and explaining risk help both of team members and customers understand the consequence of change.

vii. **Empowerment**: The Agile Development Framework also pays attention to the team working atmosphere. It is very important to set up a well organized and positive team. Therefore, encouraging team members is highly required.

b) Features of the Agile Development Framework

i. **Iterative and Incremental development**: As Figure 1.2 shows, plans, requirements, design, implementation, deployment and testing are developed incrementally through iterations. The each iteration usually takes two to four weeks. Moreover, problem hunting, scope solving, feedback collection should be done at the end of each iteration. In addition, features and tasks are also inspected and tracked within the each iteration [10].

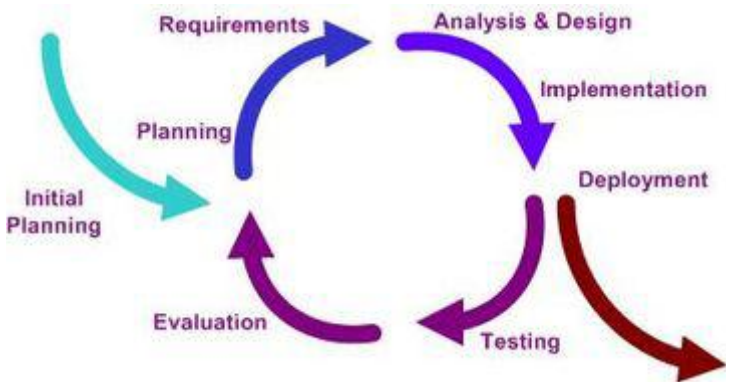

Figure 1.2: Iteration in the Agile Development Framework [11].

ii. **Establish and Changing Requirements:** In traditional software development process model, identification and analysis of requirements for the system are documented within an agreement for customers and the development team in an early phase of a project. Once this agreement has been reached, the requirements are not allowed to change. In contrast, the Agile Development Framework allows both customers and developers to change the requirements throughout the project, but only the customers have the authority to approve, disapprove and prioritize the ever changing requirements [11], refer figure 1.3. In addition, the requirements can be re-prioritized anytime. Once the priority changed, the new higher requirement will be pulled up to the top of the stack [13], refer figure 1.4.

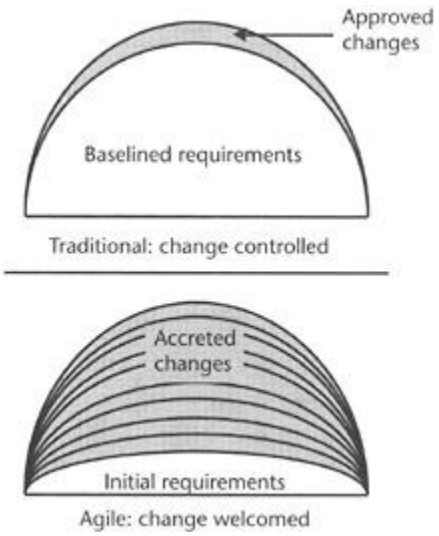

Figure 1.3: Difference of requirement change between traditional process model and the Agile Software Development [9].

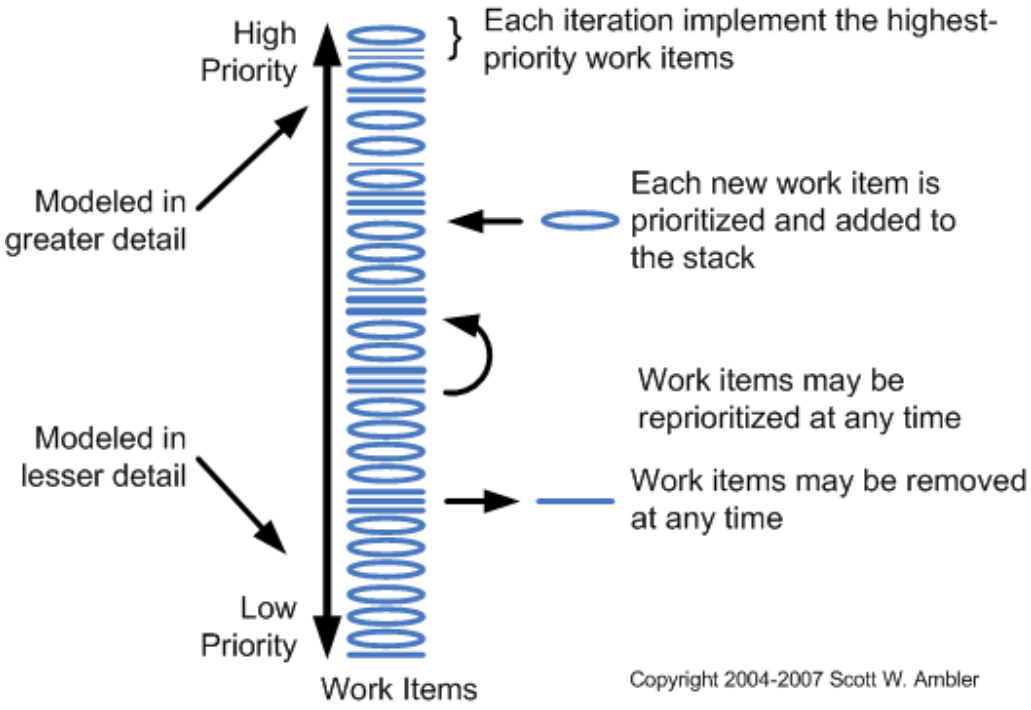

Figure 1.4: Process of Agile requirements change management [13].

iii. **Use Backlog**: In the Agile Development Framework, tasks are divided into small chunks (also known as backlogs) to manage complexity and to get quick feedback.

iv. **Prioritize Backlog**: In an Agile team, backlogs are prioritized by customers and only higher prioritized backlogs (top 2 or 3) will be processed in iteration (refer Figure 1.5). Once the backlog prioritizing works are done, the prerequisites for calling the first Sprint Planning Meeting will be embraced [14]. Moreover, unfinished backlogs are inspected and reprioritized at end of the each iteration in order to decide which backlogs will be processed in the next iteration.

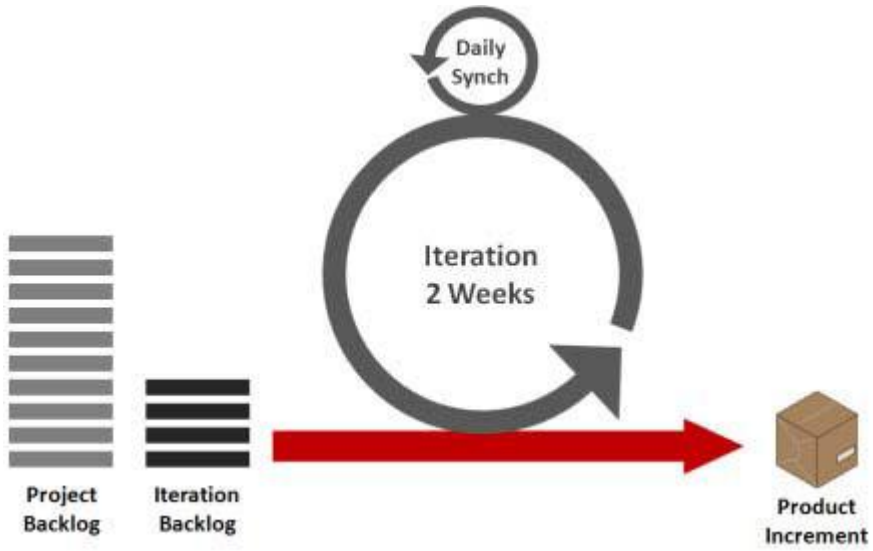

Figure 1.5: Agile Backlog Interfaces [15].

v. **Pairing Programmer & Self Organization**: In an Agile team, a less experienced programmer is paired with an experienced one in order to share knowledge and teach each other. Moreover, each agile team is self organized. Team members are self-organized by accomplishing tasks with their co-works from backlogs.

vi. **Face to Face Communication**: The Agile Development Framework emphasizes face to face communication. It promotes holding a short daily meeting. In the meeting, team members have to report their project progress. Questions such as what they have done, what they are working on, and what they will do tomorrow should be answered in daily meeting.

1.1.3 Importance of Agile Software Development

Differing from the traditional process model emphasizing the measurement of success of conformity to predictive plans, the Agile Software Development emphasizes responsiveness to change. For example, the delivery of working software is the most important factor to lead a software development successful since in the Agile Development's view, metrics such as cost variance, schedule variance, requirements variance and task variance is virtually meaningless [12]. According to the result of the DDJ 7 2007 Agile Adoption Survey [16], the Agile Development Framework has become a mainstream for software development. The survey indicates that 69% of respondents said that organizations were doing one or more Agile projects and 85% of them were even doing more than two.

Additionally, the 3rd Annual Survey 2008 [17], The State of Agile Development, shows that the users using the Agile Development Framework thought that "Accelerate time to make" and "Enhanced ability to manage changing priorities" were the top two main reasons that they were concerned about adopting the Agile Development Framework. Moreover, both of DDJ 2007 Agile adoption Survey (refer figure 1.6) and the 3rd Annual Survey 2008, The State of Agile Development (refer figure 1.7), indicate that more than 50% respondents thought that they have had 90% to 100% of successful Agile projects.

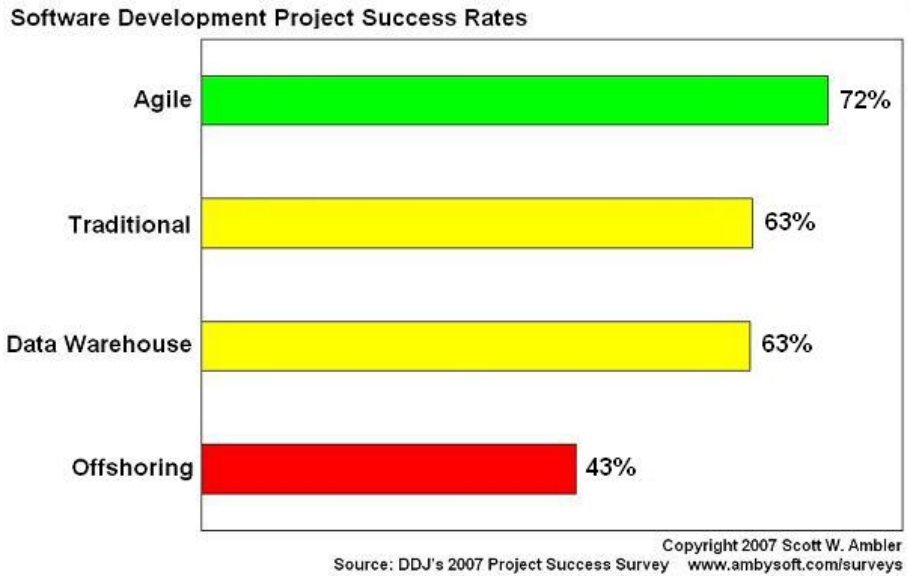

Figure 1.6: DDJ 2007 Agile Adoption Result: Rate of Successful Agile projects [16].

Successful Projects	Respondents
0%	4.8%
10%	3.7%
25%	3.8%
50%	11.5%
75%	21.2%
90% - I cant say 100% but pretty close.	37.6%
100% - I cant remember a project using Agile methods that wasn't considered successful.	17.4%

Figure 1.7: 3rd Annual Survey 2008 Result: Rate of successful agile projects [17].

Table 1.1 given below shows that almost half of respondents thought that the agile methods have improved their projects from many aspects such as project visibility, productivity, software quality, and development process *etc*.

	Significantly Improved	Improved	No Benefit	Worse	Much Worse
Improve Project Visibility	41.5%	41.8%	15.0%	1.2%	0.4%
Increase Productivity	23.6%	50.5%	15.6%	2.1%	0.3%
Enhance Software Quality	24.0%	44.3%	19.9%	2.8%	0.4%
Enhance Ability to Manage Changing Priorities	50.5%	42.1%	6.6%	0.6%	0.2%
Accelerate Time-To-Market	23.6%	41.3%	21.6%	2.4%	0.2%
Reduce Risk	16.6%	48.0%	23.6%	2.1%	0.3%

Table 1.1: 3rd Annual Survey 2008 Result: Value of implementing Agile Practices [17].

Furthermore, Figure 1.8 indicates that compared with traditional approaches, most of respondents thought that the Agile methods are more efficient.

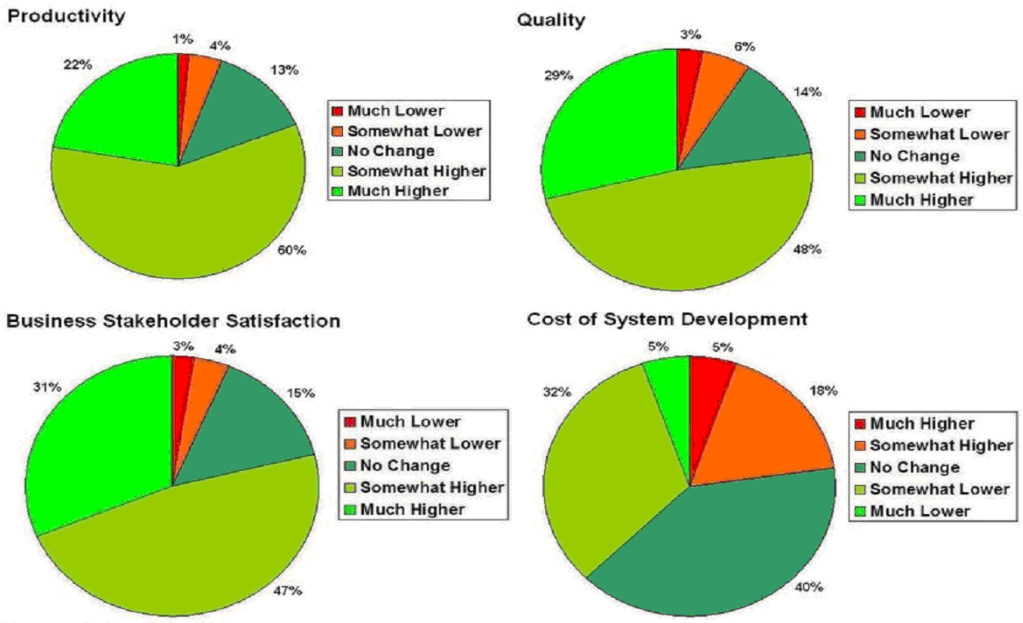

Figure 1.8: DDJ 2008 Agile Adoption Result: Comparison of effectiveness [16].

According to the survey result shown above, the Agile Development Framework is becoming a trend within software development companies. Most of respondents gave a positive view to the Agile Development Framework [18]. Moreover, comparing with traditional approaches, an agile project has more possibilities to success. Consequently, the Agile Development Framework brings software projects a great success because of the shorter feedback cycles.

For instance, Figure 1.9 shows the relationship between the cost and feedback cycle. Due to the fact that the Agile Development Framework requires a daily meeting for a frequent feedback and delivery, defects are detected in the early stage; thus the cost curve stays in a lower position, whereas, defects are usually found late in traditional approaches since they rely on a predictive plan.

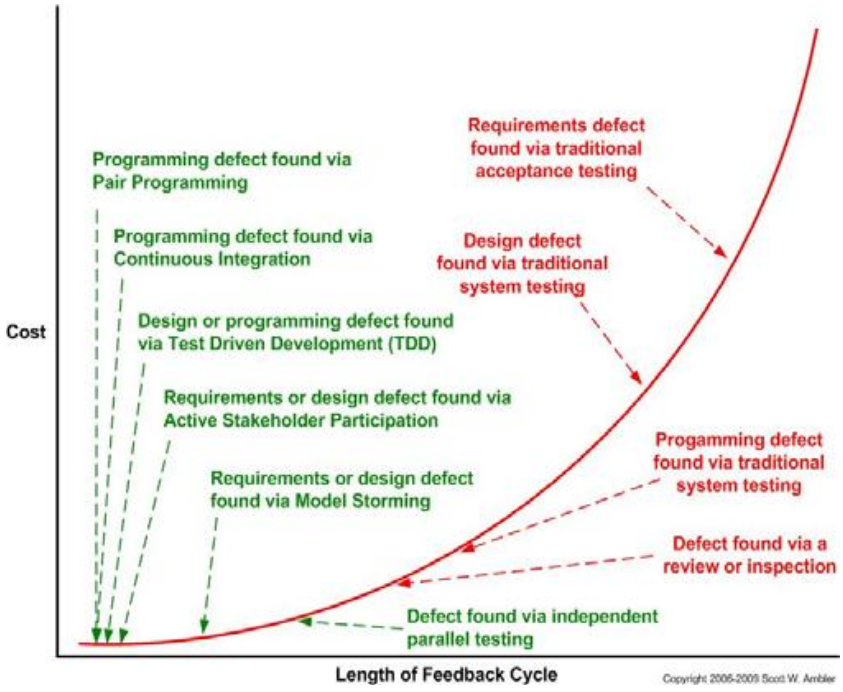

Figure 1.9: Comparison of Feedback cycles with traditional approaches [22].

1.1.4 The Agile Project Lifecycle

The Agile Development Framework is an iterative, incremental, and collaborative methodology for software development project. The Figure 1.10 shows that the Agile Software Development Lifecycle (ASDL) starts from an initial plan and ends with deployment. The each iteration consists of planning, requirements, analysis and design, implementation, deployment, testing, and evaluation.

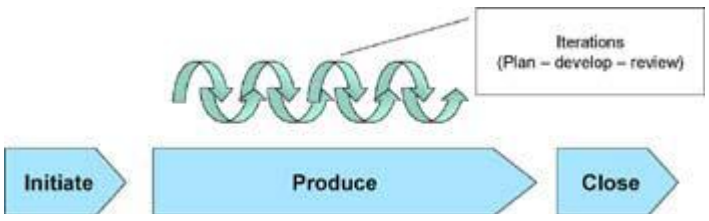

Figure 1.10: Agile Project Life Cycle [13].

The ASDL is comprised of six phases: Iteration -1, Iteration 0 (Warm up), Construction, Release (End Game), Production, and Retirement [13] refers Figure 1.11.

Description of each Phase

1. **Iteration 1**: Iteration 1 is devoted to pre project planning. Activities such as defining the business opportunity, identifying and assessing the possibility for the project are involved into this phase.

2. **Iteration 0**: After completing the iteration 1, the environment setup, team formulation, support and funding obtainment, and so on taken in the iteration 0, also known as project initiation.

3. **Construction iteration**: During construction iteration, a highly collaboration between team members and customers is required. Moreover, works regarding prioritized functionality implementation, system analysis and design, regular working software delivery, and verification are also embraced into this phase.

4. **Release**: In this phase, activities such as testing, documentation finalization, and system deployment into production have to be done in this phase.

5. **Production**: When a project turns into production phase, the goal is set as keeping systems useful and productive after deploying them to the user community. In other words, the system should be kept running and help users to use it.

6. **Retirement**: In the end of the life cycle, the iteration goes to retirement phase which is to update enterprise models and remove the final version of the system data conversion if the system has become obsolete or can be complete replaced .

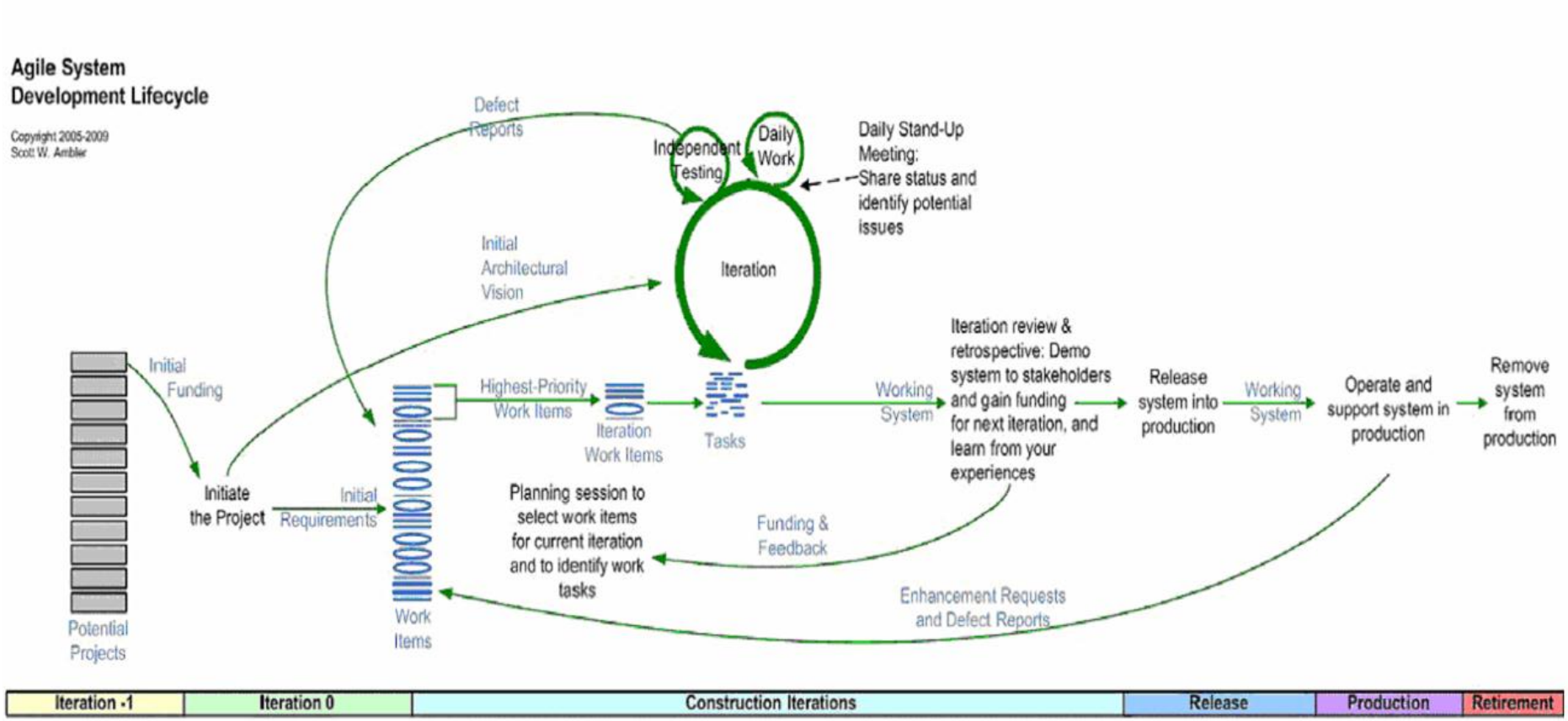

Figure 1.11: ASDL Diagram[13].

1.1.5 Methods of the Agile Development Framework

Some of the most used Agile Methods are introduced briefly in this section.

a) Adaptive Software Development (ASD)

ASD is a software development process that grew out of rapid application development work by Jim Highsmith and Sam Bayer. ASD focuses on continuous learning and adaptation to the emergent state of the project provided by a repeating series of speculate, collaborate and learn cycles. An ASD life cycle is divided into three parts: Speculate, Collaborate, and Learn and based its six characteristics which are Mission Focused, Component Based, Iterative, Time boxed, risk Driven, and Change Tolerant [23].

b) Dynamic System Development Method (DSDM)

DSDM was created in the mid 1990s. The DSDM has its roots in an iterative incremental process model that uses prototyping at each stage of development, also known as Rapid Application Development (RAD). A DSDM project is worked based on its nine principles:

- **i.** Active user involvement
- **ii.** Addressing business needs
- **iii.** Base-lining of high-level scope
- **iv.** Communication and collaboration among all stakeholders
- **v.** Frequent delivery
- **vi.** Team decision making
- **vii.** Integrated testing
- **viii.** Iterative-incremental development
- **ix.** Reversible changes throughout development

c) Extreme Programming (XP)

XP was developed by Ken Beck, Ward Cunningham and Rom Jeffries during the 1990s. XP is one of the most adapted Agile Methods in the high technology industry. The XP highlights the feature of pair programming and test driven development. Moreover, it is often seen as an agile engineering process although it provides planning practices for project management [16].

d) Feature Driven Development (FDD)

FDD is a client centric, architecture centric and pragmatic software process. It was first introduced in 1999 via the book Java Modeling in Color with UML. Differing from other the Agile Methods, the FDD was first applied in an 18 month long middle-large project containing 50 persons [15]. A FDD life cycle includes the following phases: Develop an Overall Model, Build a Features List, Plan by Feature, Design by Feature, and Build by Feature [20].

e) Lean Software Development (LSD)

LSD, created by Bob Charette, is the application of lean principles to the craft of software development. LSD is a strategy oriented approach that considers the expenditure of resources for any goal. Software development is based on the goals of one third the time, one third the budget, and one-third the defect rate [18].

The LSD principles stated by Norton (2005) are:

i. Eliminate Waste
ii. Amplify Learning
iii. Decide as late as possible
iv. Deliver as fast as possible
v. Empower the team
vi. Build integrity in
vii. See the whole

f) SCRUM

Same as XP, SCRUM is one of the most used Agile Methods. It is a lean approach to software development. In SCRUM, development is structured in iterative cycles of work called Sprint. An Iteration of work usually takes 2 to 4 weeks. Team works on prioritized tasks by customers during the each iteration and at the end of each Sprint, a potentially shippable product should be delivered. SCRUM was created by Ken Schwaber and Jeff Sutherland in the 1990s. SCRUM is usually used as an agile project management method rather than an agile process [21]. A short daily meeting in order to give team members a quick update is required in the SCRUM framework [19].

g) IBM Rational Unified Process (RUP)

RUP is also listed as one of the Agile Methods in the work of Krebs [22]. RUP was invented by IBM in 2003. It provides industry tested practices a comprehensive process framework for software and systems delivery and implementation for effect project management. In RUP, users can customize the project process to meet their unique demands by selecting and deploying the process elements they need for their projects.

1.2 Challenges to Agile Processes

This should not be surprising; none of the agile processes is a silver bullet (despite the enthusiastic claims of some its proponents). In this part some of the situations in which agile processes may generally not be applicable are given. It is possible that some agile processes fit these assumptions better, while others may be able to be extended to address the limitations discussed here. Such extensions can involve incorporating principles and practices often associated with more predictive development practices into agile processes.

a) Distributed Development Environments

The emphasis on co-location in practices advocated by agile processes does not fit well with the drive by some industries to realize globally distributed software development environments. Development environments in which team members and customers are physically distributed may not be able to accommodate the face-to face communication advocated by agile processes. In such cases, one can at least approximate face-to-face communication using technologies such as videoconferencing, but these technologies are expensive and not as effective as one would hope. Face-to-face communication is as important in distributed environments as non-distributed environment, but it occurs less frequently and has to be planned in advance to ensure that all involved can participate. One can use such face-to-face meetings as major synchronization events in which geographically dispersed developers (1) are made aware of the progress made by others and (2) discuss plans for further evolving the product. In between such meetings, documentation (beyond code) becomes the primary form of communication. Good documentation of

requirements and designs, produced and maintained in a timely manner, are essential to ensure that the distributed team members all maintain the same vision of the product to be built. This should not be interpreted as a requirement to document or model all aspects of software. Documentation and models should be created and maintained only if they provide value to the project and project stakeholders.

b) Subcontracting

Outsourcing of software development tasks to subcontractors is often based on contracts that precisely stipulate what is required of the subcontractor. Subcontracted tasks have to be well-defined in the cases where subcontractors have to bid for the contract. In developing a bid a subcontractor will usually develop a plan that includes a process, with milestones and deliverables, in sufficient detail to determine a cost estimate. The process may be an iterative, incremental approach, but the subcontractor may have to make the process predictive by specifying the number of iterations and the deliverables of the each iteration in order to compete. It is possible that a contract can be written that allows a subcontractor some degree of flexibility in how they develop the product within time and cost constraints [24]. This is certainly possible if the subcontractor has a good track record and can be trusted by the contracting company to develop a product that meets the contracting company's needs.

A contract supporting agile development in the subcontractor environment should consist of two parts:

(1) Fixed Part: This part defines

i. The framework that constrains how the subcontractor will incorporate changes into the product (e.g., cost- and time-based criteria for accepting or rejecting changes to the Variable Part (see below) of the contract.
ii. The activities that must be carried out by the subcontractor (e.g., quality assurance activities).
iii. Requirements that are to be considered fixed and deliverables that must be delivered.

(2) Variable Part: This part defines the requirements and deliverables that can vary within the boundaries defined in the Fixed Part. This part can evolve within the

constraints defined in the Fixed Part. At the time the Contract is signed; a description of prioritized deliverables and requirements should be included.

c) Reusable Artifact Development

Agile processes such as Extreme Programming focus on building software products that solve a specific problem. Development in "Internet time" often precludes developing generalized solutions even when it is clear that this could yield long-term benefits. In such an environment, the development of generalized solutions and other forms of reusable software (e.g., design frameworks) is best tackled in projects that are primarily concerned with the development of reusable artifacts. This separation of the product-specific development environment from the reusable artifact development environment is a primary feature of the reuse-oriented framework called the Experience Factory developed by researchers at the University of Maryland at College Park [26]. The wide applicability of a reusable artifact requires that the process used to build the artifact emphasize quality control because the impact of low quality (in particular, severe errors) is as wide as the number of applications that reuse the artifact. On the other hand, timely development of reusable artifacts is desirable. While there seems to be a case for applying agile processes to the development of reusable artifacts, it is not clear how agile processes can be suitably adapted.

d) Development involving Large Teams

Agile processes support process "management-in-the small" in that the coordination, control, and communication mechanisms used are applicable to small to medium sized teams. With larger teams, the number of communication lines that have to be maintained can reduce the effectiveness of practices such as informal face-to-face communications and review meetings. Large teams require less agile approaches to tackle issues particular to "management-in-the-large". Traditional software engineering practices that emphasize documentation, change control and architecture-centric development are more applicable here. This is not to say that agile practices are not applicable in such environments. There may be opportunities for teams to use agile practices, but the degree of agility possible may be less than that found in smaller projects.

e) Developing Safety-Critical Software

Safety-critical software is software in which failure can result in direct injury to humans or cause severe economic damage. The quality control mechanisms supported by current agile processes (e.g., informal reviews, pair programming) have not proven to be adequate to assure users that the product is safe. In fact there is some doubt that these techniques alone will be sufficient. Formal specification, rigorous test coverage, and other formal analysis and evaluation techniques included in software engineering approaches provide better, but also more expensive, mechanisms to tackle the development of safety- or business-critical software. Some agile practices can also bring benefits to the development of such software. For example,

a) Test-first approaches require one to define unit tests before writing code.

b) The early production of working code supported by the incremental, iterative process structure of agile processes supports exploratory development of critical software in which requirements are not well-defined.

c) Pair-programming can be an effective supplement to formal reviews. Therefore, it can be assumed that agile and formal software development are not incompatible, but can be combined when needed: Formal techniques may be used in an agile way to handle critical pieces of the software to increase quality and confidence.

f) Developing Large, Complex Software

The assumption that code refactoring removes the need to design for change may not hold for large complex systems in particular. In such software, there may be critical architectural aspects that are difficult to change because of the critical role they play in the core services offered by the system. In such cases, the cost of changing these aspects can be very high and therefore it pays to make extra efforts to anticipate such changes early. The reliance on code refactoring could also be problematic for such systems. The complexity and size of such software may make strict code refactoring costly and error-prone. Models can play an important role here, especially if tools exist for generating significant portions of the code from the models. This view of models as the central artifacts for evolving systems is at the heart of the Object

Management Group's (OMG) Model-Driven Architecture (MDA) approach [40]. There may also be systems in which functionality is so tightly coupled and integrated that it may not be possible to develop the software incrementally. In these cases an iterative approach in which code is produced in the each iteration can still be used, but the code produced in the each iteration will include all the pieces in various states of incompleteness.

1.3 Difference in Agile and Heavyweight Methodologies

Traditional development approaches have been around for a very long time. Since its introduction the waterfall model has been widely used in both large and small software projects and has been reported to be successful to many projects. Despite the success it has a lot of drawbacks, like linearity, inflexibility in changing requirements, and high formal processes irrespective of the size of the project. Kent Beck took these drawbacks into account and introduced Extreme Programming, the first agile methodology produced. Agile methods deal with unstable and volatile requirements by using a number of techniques, focusing on collaboration between developers and customers and support early product delivery. A summary of the difference of agile and heavyweight methodologies is shown in the table 1.2 given below.

	Agile Methods	Heavy Methods
Approach	Adaptive	Predictive
Success Measurement	Business Value	Conformation to plan
Project size	Small	Large
Management Style	Decentralized	Autocratic
Perspective to Change	Change Adaptability	Change Sustainability
Culture	Leadership-Collaboration	Command-Control
Documentation	Low	Heavy
Emphasis	People-Oriented	Process-Oriented
Cycles	Numerous	Limited
Domain	Unpredictable/Exploratory	Predictable
Upfront Planning	Minimal	Comprehensive
Return on Investment	Early in Project	End of Project
Team Size	Small/Creative	Large

Table 1.2: Difference in Agile and Heavyweight Methodologies [26].

The agile and heavyweight methodologies both have their strengths and weaknesses. People usually follow either one of these methodologies or follow their own customized methodology. There are major factors affecting methodology decision and

selecting which is suitable for various conditions. These factors can be categorized into project size, people and risk [26].

1.4 Reusability verses Agile Software Development

Reusability comes with using the legacy software's source code and other software development assets in the under-development projects or in the future projects. To achieve the Reusability there should be proper documentation and design of the system required. It is very difficult to understand the code of the legacy system without proper documentation and design. Also it is not always possible to use a piece of code as such, it may need to modified according to the requirements and without the proper understanding of the code structure modification can never occur successfully and seamlessly. As per in case of Agile there is no such concept of reusability. The soul purpose behind using the agile process for software development is quick and fast delivery. There is no proper planning, design and documentation of the software has been done. If planning, design and documentation are embedded in agile software development process then it'll not possible to develop a project without a delay because planning, design and documentation consume a lot of time. So Reusability requires proper documentation and planning but Agile says software is nothing but code. Reuse does not just happen; it needs to be planned.

1.5 Organization of Thesis

Further chapters of thesis are organized as below:

Chapter 2 Reviews Literature on Reusability, Reusability operations, Approaches, Reusable Assets of Software Development and Review of Reverse Engineering methods, types of Reverse Engineering. Moreover, UML tools for Reverse Engineering have been explored.

Chapter 3 concentrates on problem statement i.e. what is the actual problem and how it can be solved. Here objectives of thesis are identified.

In Chapter 4, Agile software development model using reverse engineering is proposed in order to add flavor of reusability in agile environment.

In Chapter 5, the Proposed Model is implemented using a case study.

The Chapter 6 Concludes and defines the future scope of the work.

Chapter 2
Literature Survey

2.1 Reusability

Reusability means using a segment of source code that can be used again to add new functionalities with slight or no modification. In most engineering disciplines, systems are designed by composing existing components that have been used in other systems. Software engineering has been more focused on original development but it is now recognized that to achieve better software, more quickly and at lower cost, we need to adopt a design process that is based on systematic software reuse [25].

2.1.1 Reuse-based software engineering

Different type of reusability given below:

a) Application system reuse

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

b) Component reuse

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

c) Object and function reuse

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

2.1.2 Incremental Stages of Reuse

The different reuse approaches and reusable assets by presenting an incremental view. The view is in line with the results revealed by the initial interviews held for the problem analysis. With this view it remains important to note that the highest levels of reuse are not necessary the best, this rather depends on other factors such as the scope of the domain and the development strategy. An overview of the incremental levels is presented in Figure 2.1.

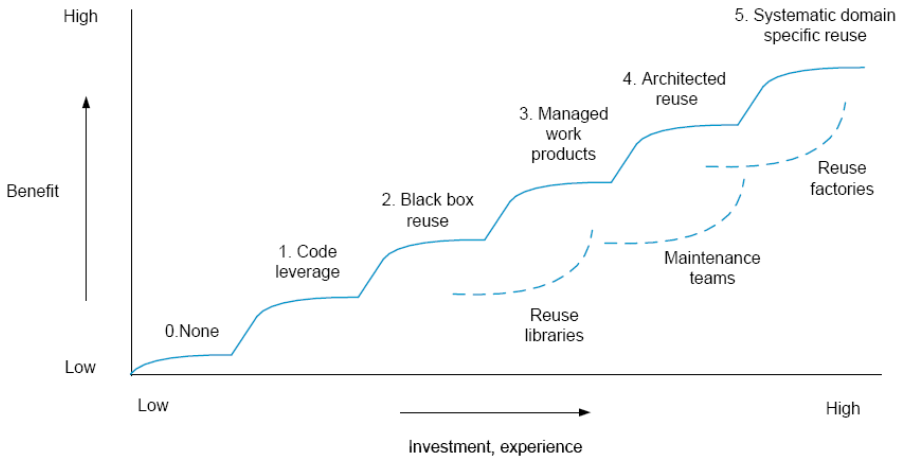

Figure 2.1: Incremental stages of reuse [27].

Figure 2.1 summarizes incremental reuse levels (solid line) and related reuse approaches (dotted lines). The reuse of ad-hoc reuse events with initial benefits is the only level of reuse which can be achieved without investing in software reuse; instead experience of previous projects is used to copy relevant pieces of code. This reuse level is defined as level 0. The first real reuse level presents a form of code-leverage, where pieces of code are made available and can be reused by multiple parties. The pieces of code are made available through the use of a reuse library, providing a central place where the components are stored. The use of these leveraged components is not restricted and can be considered white-box reuse; the code may be adjusted and changed to the specific context in which the component is applied. This strategy works for a while up to then point that multiple copies, each slightly different, have to be managed. The choice can be made to stop using components as white-box and start using them as black-box components instead. Black-box components may no longer be internally adjusted; rather its environment should be adjusted to support the component.

Previous research has pointed out that with black-box reuse higher reuse levels can be achieved than with white-box reuse. The reason for this is that there are reduced costs in component maintenance and maintenance across products using these components. However, black-box reuse also has its limitations. Systematic software reuse is presented in literature as the solution to achieve higher reuse levels [28]. This

approach is strengthened by various researchers reporting both productivity and quality gains with more formalized processes [29]. Such an approach is often rather static, where the assumption is made that components are predefined and carefully tested before they are populated into a repository. Only a few researchers noted the dynamic aspects of software reuse. Henninger is one of them and he notes that: repositories for software reuse are faced with two interrelated problems: acquiring the knowledge to initially construct the repository, and modifying the repository to meet the evolving and dynamic needs of software development organizations' [30]. When taking a product line approach, additional insights regarding software component evolution are gained. Tomer et al. describe the evolving dynamic aspects along three axes: the development, maintenance and reuse axis [31]. An overview of this model is presented in Figure 2.2.

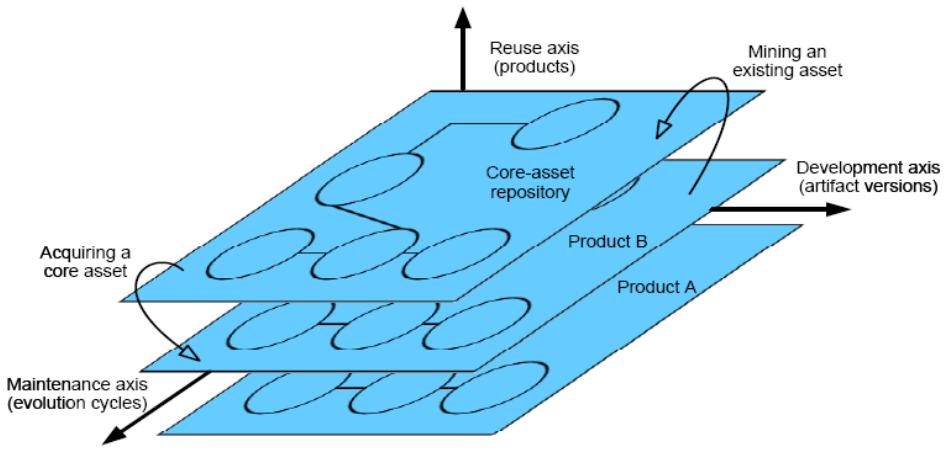

Figure 2.2: The three-dimensional evolution of a software product line [31].

The axis of the three dimensional model are explained by an underlying model, also presented in Tomer et al. [31]. The underlying model is presented in Figure 2.3. For simplicity Tomer et al. transferred the underlying model to a two dimensional model, where development and maintenance are combined into one axis. A key assumption made in this model is that reuse activities cannot be freely transferred between specific products without first storing and cataloguing the assets in a central repository [31].As mentioned earlier, reusable software assets can be used as black-box or white-box components [32]. The model of Tomer et al. [31] explicitly

recognizes this distinction. With black-box the component is used 'as-is', which means that it is not adjusted, but instead its environment has to be adjusted to it. With white-box reuse the component itself is adjusted, allowing multiple versions of the same component simultaneously. Higher reuse gains are reported with black-box reuse, which is in line which Krueger's theory of reuse and abstractions. According to this theory reuse can only be successful when abstractions are present hiding the internal logic of components [33].

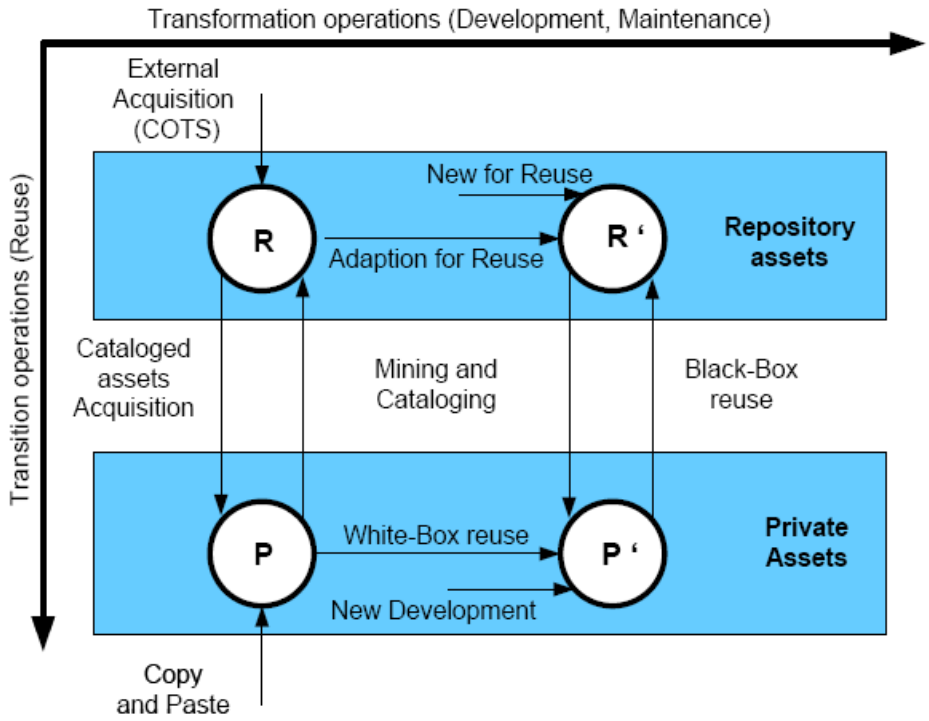

Figure 2.3: Assets and reuse operations [31].

The key message of software reuse is that the costs related to the reusable assets are lower than when all components are redeveloped from the ground up. This means that with a successful reuse program the average cost of searching for a component, determining its reusability and adjusting or building a new one is lower than the costs of continuously building new components [33]. In practice also indirect costs and gains play a role, such as the possible shorter time to market and the scalability of applications [34].

2.1.3 Reusable Assets of Software Development

Following are the various software assets which can be re-used with little or no modification in the future projects.

i. Architecture
ii. Source Code
iii. Data
iv. Designs
v. Documentation
vi. Templates
vii. Human Interfaces
viii. Plans
ix. Requirements
x. Test Cases

2.1.4 Approaches to Software Reuse

Although reuse is often simply thought of as the reuse of system components, there are many different approaches to reuse that may be used. Reuse is possible at a range of levels from simple functions to complete application systems. The possible reuse techniques given in figure 2.4.

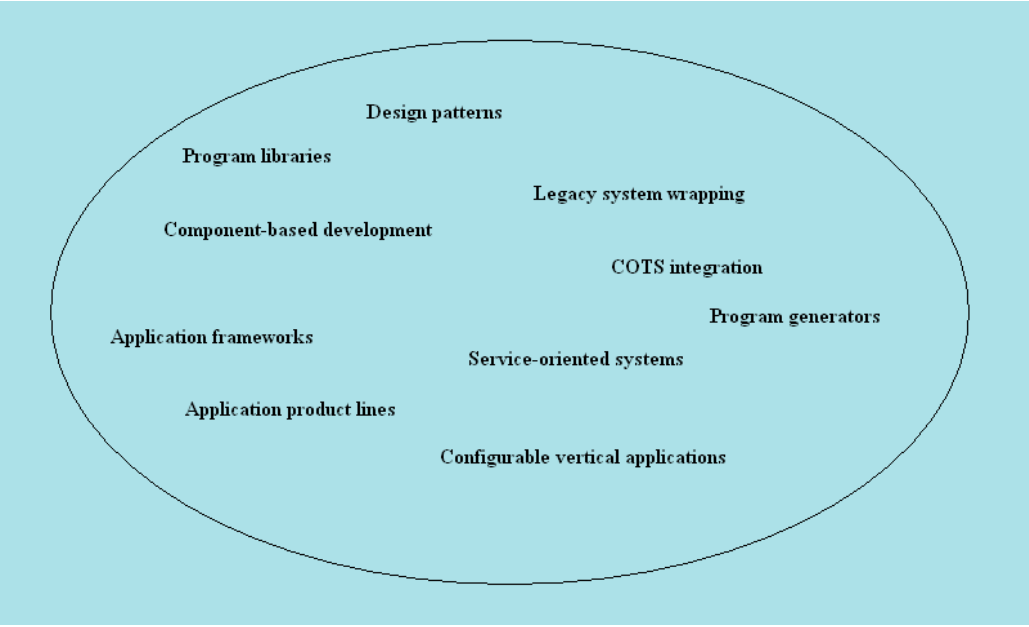

Figure 2.4: Various approaches to achieve reusability [37].

Brief explanation of the each of the Reuse Approach

a) **Design patterns**

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

b) **Component-based development**

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

c) **Application frameworks**

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

d) **Legacy system wrapping**

Legacy systems that can be 'wrapped' by defining a set of interfaces and providing access to these legacy systems through these interfaces.

e) **Service-oriented systems**

Systems are developed by linking shared services that may be externally provided.

f) **Application product lines**

An application type is generalized around a common architecture so that it can be adapted in different ways for different customers.

g) **COTS integration**

Systems are developed by integrating existing application systems.

h) **Configurable vertical applications**

A generic system is designed so that it can be configured to the needs of specific system customers.

i) **Program libraries**

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

j) Program generators

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

2.2 Reverse Engineering

Reverse engineering means evaluating something to understand how it works in order to duplicate or enhance it. It allows the reuse of the know-how hidden in already implemented programs. The object oriented software developers now admit that thinking about object-oriented program understanding and comprehension to be relatively easier is not that easy. Programs are even more complex and difficult to comprehend, unless rigorously documented. What if the documentation is improper? To affect change management, even a simpler upgrade may become cumbersome then. This is the reason why eminent development houses now focusing on advanced documentation support. Re-engineering code environment thence largely affect the problem issues regarding program comprehension when the software size grows enormously. Reverse Engineering is a methodology that greatly reduces the time, effort and complexity involved in solving these issues providing efficient program understanding as an integral constituent of re-engineering paradigm.

Reverse engineering is a systematic form of program understanding that takes a program and constructs a high-level representation useful for documentation, maintenance, or reuse. To accomplish this, reverse engineering technique begin by analyzing a program's structure. The structure is determined by lexical, syntactic, and semantic rules for legal program construction. Because it is known how to proceed on these kinds of analysis, it is natural to try and apply them to understanding programs. Initially reverse engineering term was evolved in the context of legacy software support but now has ventured into important issue of code security such that it doesn't remain confined to legacy systems. Transformations are applied under the process of Re-engineering after analyzing the software to apply changes incorporating new features and provide support for latest environment. Object-oriented software development methodology primarily has three phases of Analysis, Design and Implementation [35]. With the view of traditional waterfall model, reverse engineering thus is looking back to design from implementation and to analysis from

implementation. Important thing is that it actually is a reverse forward engineering i.e. from implementation; analysis is not reached before design. Simple schematic diagram to elaborate reverse engineering is shown in Figure 2.5.

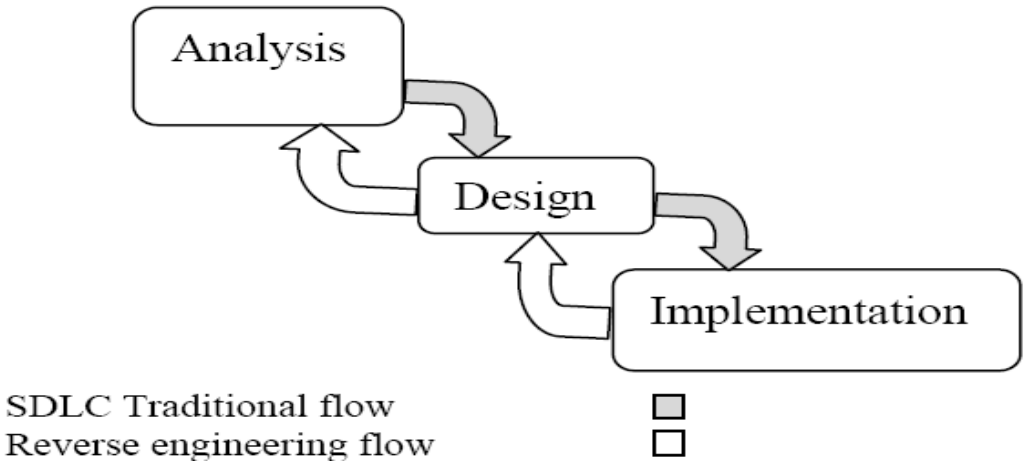

Figure 2.5: A simple representation to reverse engineering of object-oriented development [35].

It is the process of analysis. The software system or program under study is neither modified nor re-implemented because of not bringing it under Re-engineering. [36] Software Re-engineering is the area which deals with modifying software to efficiently adapt new changes that can be incorporated within as software aging is a well known issue. Reverse engineering provided cost effective solution for modifying software or programs to adapt change management through Re-engineering application.

2.2.1 Importance of Reverse Engineering

Reverse engineering has made its place and importance in the software engineering field because of the following advantages provided by it:

i. It helps in Modifying software.

ii. Helps in Design and implementation in forward engineering, e.g. debugging.

iii. It helps to understanding the program structure.

iv. Reverse engineering provides help in Program visualisation.

v. Software re-usability can be achieved using reverse engineering.

2.2.2 An Insight on Reverse Engineering

Reverse Engineering Techniques can be classified in two ways:

a) Programmer's view

It is based on the input to be provided to the tools or environments to analyze the system under study. A programmer has prime concern for three aspects- creational, structural and behavioral. Usually that affects two types of reverse engineering paradigms-

i. **Code Reverse Engineering:** In this case, there is no source code available for the software, and any efforts towards discovering one possible source code for the software are regarded as reverse engineering. This second usage of the term is the one most people are familiar with. Mainly the structural analysis methodology is undertaken for code reverse engineering.

ii. **Data Reverse Engineering**: In this case, source code is already available for the software, but higher-level aspects of the program, perhaps poorly documented or documented but no longer valid, are to be discovered. The analysis technique involved in such type of reverse engineering is sometimes called Reverse Forward Engineering. Thread based analysis approaches are useful for data reverse engineering.

b) Analyzer's view

It is based on the type of analysis conducted on software or program to inspect the properties to look into structural or behavioral aspects of the system under study. Two major kinds of analysis can be done under reverse engineering- Structural Analysis and Behavioral Analysis. The former is called static analysis whereas the later one is known as Dynamic Analysis. Java is an object oriented language centered on Objects reflecting real time behavior. Structural analysis of java code can be used to identify code elements viz. Attributes, fields, methods and other code artifacts whereas Behavioral analysis is concerned with the runtime behavior of objects created and then garbage collected during the program execution. These two kinds of analysis under analyzer's view are described further. However we have put stress on static analysis only.

i. **Static Analysis:** Static analysis deals with the program understanding at syntactic level hence static reverse engineering technique is used for obtaining static structural artifacts from intermediate code. Java bytecode has a specification popularly known as JSR document, specifying the programming language alphabet set and grammar rules to produce those symbols into a syntactically correct program through mnemonics. Thus static analysis of the source code may be carried out with the same initial methods that are used for compilation of bytecodes. Parsing the source code, enables to extract some of the basic code details such as the classes, the inheritance hierarchy, the API' s, the variables declared etc. Since Java is object-oriented, thence it has a class structure, which holds some object(s) which have data or group of data, and also each class has methods that can be used to access and manipulate the data contained within.

ii. **Dynamic Analysis:** Runtime Dynamic analysis of java code is required for analyzing features related to multithreaded program support, analyzing behavioral aspects of memory leakage, CPU utilization and other aspects. Our study is focused on decision for sufficiency of static analysis to mine structural code from the bytecode since intermediate bytecode contains enough information about code elements.

2.2.3 UML Reverse Engineering Process

The UML reverse engineering tool U-Model generates class diagrams from class files as well as source files, whereas Enterprise Architect takes archive input as shown in figure 2.6.

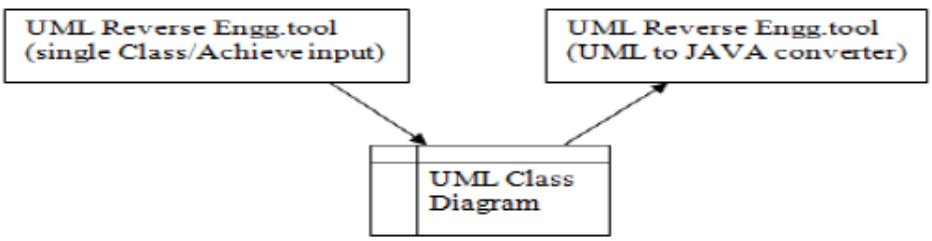

Figure 2.6: Block diagram for UML Reverse Engineering Process [39].

The procedure for reverse engineering through program analysis is by using these tools and manual comparison of class diagrams from original vs. decompiled source code. The extent to which the output from these tests are approximately same, defines the sufficiency of structural code recovery using static analysis [39].

2.2.4 UML Reverse Engineering Tools

UML reverse engineering tools are capable of converting source or class files into class design from which original source can be coded. UML reverse engineering tools provide design phase information of development cycle from the code for an object oriented software. These tools are also of two kinds primarily based upon UML Reverse Engineering Process –

1. **Unidirectional Reverse Engineering tools***:* Those which take class files or achieve files (.jar) as input to produce class diagrams (UML models).

2. **Round Trip Engineering Tools***:* Some tools facilitate round trip engineering either starting from source code or Binary bytecode with UML model as an intermediary. Apart from above, some tools only take source file as input and produce UML class diagrams and vice versa. These are not actually Reverse Engineering tool but referring to the first chapter , going back in development cycle make them eligible to be called as Reverse Engineering tool. Also these can produce equivalent sequence diagrams for runtime analysis, or collaboration diagrams.

Partial list of Tools available for OO languages

- Produce a class diagram from code
  - Rational Rose (Rational Software Corp.)
  - Paradigm Plus (Computer Associates International)
  - OEW (Innovative Software GmbH)
  - Graphical Designer (Aadvanced Software Technologies Inc.)
  - Many more....

2.3 Conclusion

From the overall literature survey, it can be concluded that different things: code, design, test cases etc can be reused. Reuse can be systematic (software development for reuse), or opportunistic (software development with reuse) Reuse does not just happen; it needs to be planned and require proper documentation and design. Reverse-engineering can be used for reusability or it can be said that reusability can be achieved using reverse-engineering. Reverse engineering helps to understand the legacy system by creating its UML model and once the model of the legacy system is created, that model can be used with little or no modification in the underdevelopment or in the future project to promote reusability and increase productivity of the organization. There are lots of UML tools available to perform reverse-engineering process.

Reverse-Engineering can be used to make the poorly designed and poorly documented legacy software system developed with agile software development process, Re-usable by extracting the component from the source code of the legacy system using UML models . Reuse based software engineering and agile development is an open research area in rapid development.

Chapter 3
Problem Statement

3.1 Problem Definition

Legacy Software Systems developed with Agile Software Development are often poorly designed and documented, but still perform a good job for the organization critical application. Due to poor design and documentation it becomes very difficult to reuse the functionality of legacy systems in the future projects. The importance of the legacy system cannot be undermined because some of their functions are too valuable to be discarded and too expensive to reproduce. Be precise agile software development is favorite of all type of organization small, medium or large. But the problem being faced by the software industry is that agile software development generates specialized products which can't be reused again. This all the development effort goes waste. So the biggest challenge is to integrate reusability with the Agile Software Development Environment.

3.2 Objectives

1. To study the Agile Software Development.
2. To study Agile Software Development with respect to Reusability.
3. To study different strategies of integrating reusability in agile environment.
4. Design the solution and propose a model which can enhance reusability in Agile Software Development environment.

Chapter 4
Proposed Model

4.1 Re-engineering based Component Generation and Retrieval Model

As we have already discussed in previous chapters that Legacy software systems developed with Agile Software Development are often poorly designed and documented, but still perform a good job for the organization's critical application. Due to poor design and documentation it becomes very difficult to reuse the functionality of legacy system in the future projects. The importance of the legacy system cannot be undermined because some of their functions are too valuable to be discarded and too expensive to reproduce. Software Re-engineering with effective design and architecture can make better system with respect to reusability and maintainability. Software Re-engineering attempts to understand the existing software by reverse engineering and redesigning and scaling it up into new software as per newer demands. The main motive behind the re-engineering is integrating the legacy system with component based technologies. In component based software engineering, the software systems are developed by assembling software systems from pre-built software units or components. The motivations behind the use of components are faster development, lower cost of development and better usability.

So a model named Re-engineering based Component Generation and Retrieval model has been proposed based on the Reverse-Engineering to extract the re-usable components from the Legacy Software System, store them in a component repository and use these components in the future projects. It has two phases as given below:

1. ES Phase (Extract and Store phase): In this phase we extract the reusable components from the legacy software system's source code and store these components in the component repository.

2. SREM Phase (Search, Retrieve, Evaluate and Modify phase): In this phase components are searched, retrieved, evaluated and modified according to the requirement and used in the software development.

Detailed explanation of each of this phase is given below:

4.1.1 Extract and Store Phase

As shown in the figure 4.1 we supply the source code to a reverse engineering tool to create its design in the form of UML diagrams, its services and its documentation.

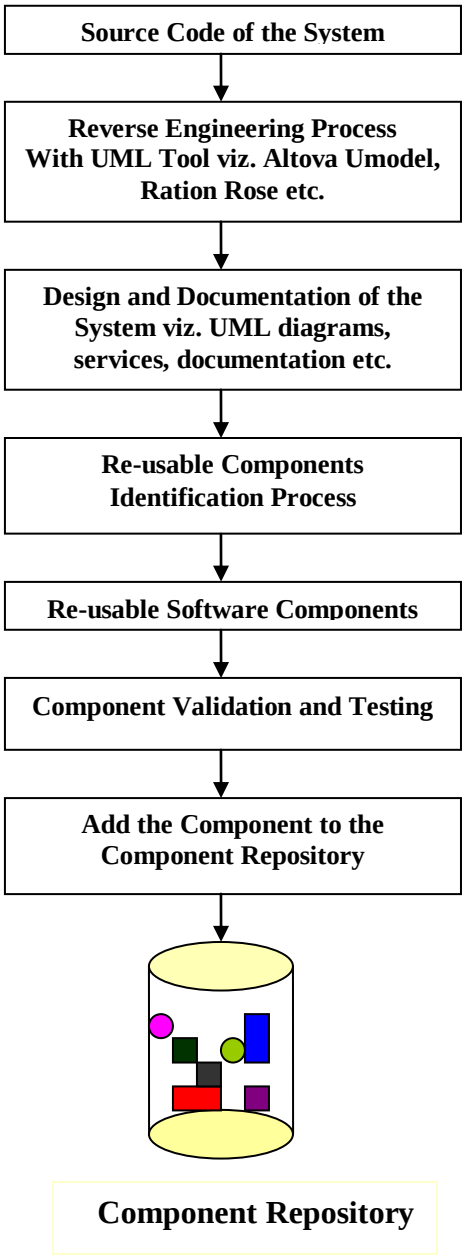

Figure 4.1: ES Phase: Extract and Store Phase

Then the re-usable component identification process is applied using Object Oriented Analysis and Design (OOAD) approach by creating the UML diagram like Use Case diagram, Interaction diagram, Component diagram etc. Then the component validation and testing is done. After the component validation and testing has completed, identified reusable components are stored in the component repository. Whole process is illustrated in pictorial form (refer figure 4.1).

a) Source code Consideration

The source code for a particular piece of software may be contained in a single file or many files (refer figure 4.2). Though the practice is uncommon, a program's source code can be written in different programming languages. For example, a program written primarily in the C programming languages might have portions written in assembly language for optimization purposes. It is also possible for some components of a piece of software to be written and compiled separately, in an arbitrary programming language, and later integrated into the software using a technique called library linking. This is the case in some languages, such as Java: each class is compiled separately into a file and linked by the interpreter at runtime.

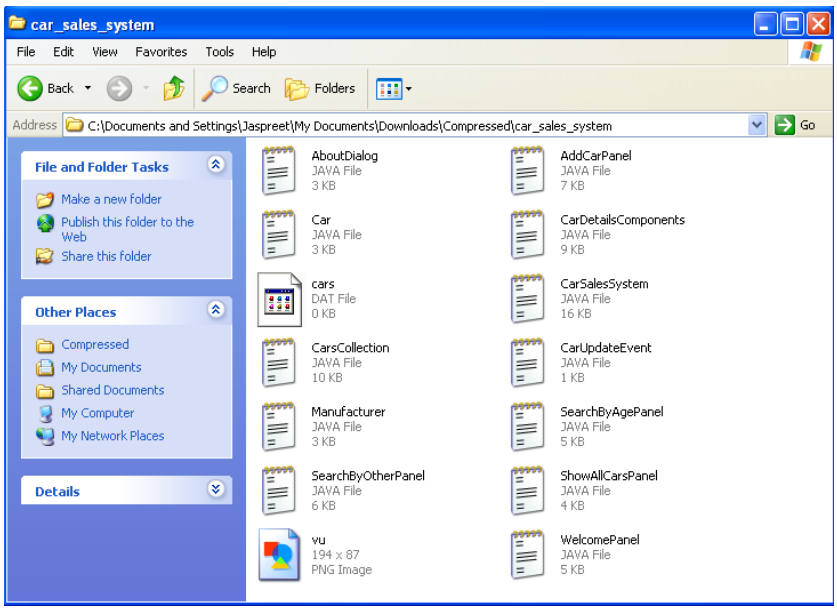

Figure 4.2: Source code of the car sales System.

b) Design with Automated Tool (UML)

Reverse engineering is the process of analyzing a subject system to create representations of the system at a higher level of abstraction." It can also be seen as "going backwards through the development cycle". A number of UML tools refer to the process of importing and analyzing source code to generate UML diagrams as "reverse engineering". Altova UModel is an intuitive, affordable, and fully featured tool to create UML use cases and additional advantages of UML-based software development. All 14 UML diagram types can be created using this tool by importing the source code of the software as shown in below figure 4.3.

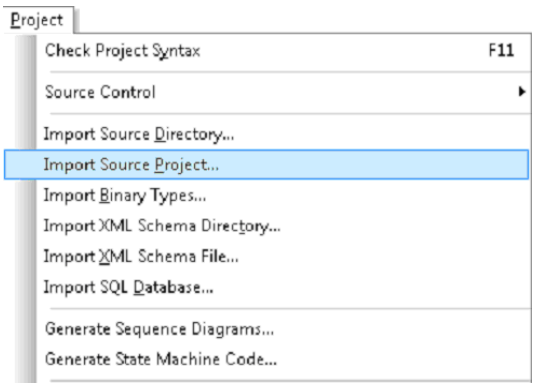

Figure 4.3: Importing the source code in Altova UModel Tool.

UModel can import Java source code files from JBuilder, Eclipse, and NetBeans projects (refer figure 4.4), C# source code from Microsoft Visual Studio and Borland C#, and Visual Basic .NET project files.

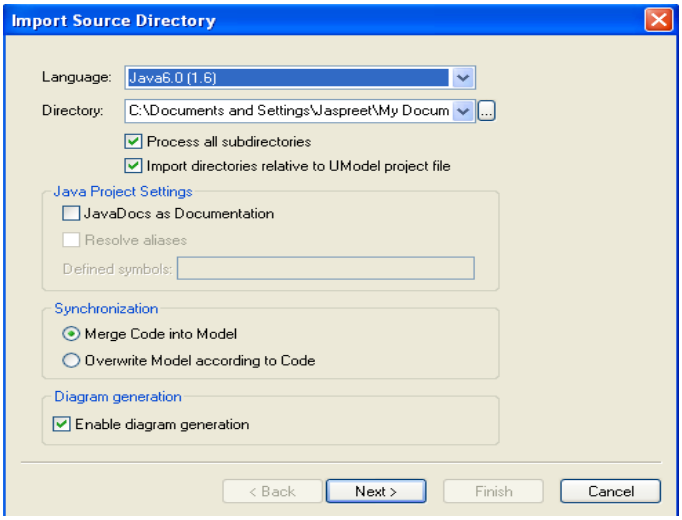

Figure 4.4: Selecting the source directory of the project.

We can import a single directory, a directory tree, or an entire project, and we can choose to merge the imported code into an existing UModel project, or create a new. We can apply reverse engineering to model an existing application. Or, we can get a new project off to a quick start by importing class libraries such as employee, customer, vendor, and other classes that are already known to our organization.

c) Design and Document of the system

The design and documentation of a software system is written description of a software product, that a software designer writes in order to give a software development team an overall guidance of the architecture of the software project. Architecture of the software can be better understood by UML diagrams (refer figure 4.5). It usually accompanies an architecture diagram with pointers to detailed feature specifications of smaller pieces of the design. Practically, a design document is required to coordinate a large team under a single vision.

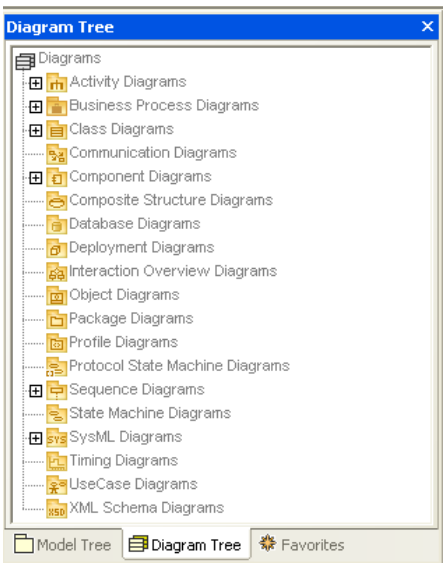

Figure 4.5: Diagram tree in Altova UModel Tool.

A design document needs to be a stable reference, outlining all parts of the software and how they will work. The document is commanded to give a fairly complete description, while maintaining a high-level view of the software. These documents do not describe how to program a particular routine, or even why that particular routine

exists in the form that it does, but instead merely lays out the general requirements that would motivate the existence of such a routine. A good architecture document is short on details but thick on explanation (refer figure 4.6).

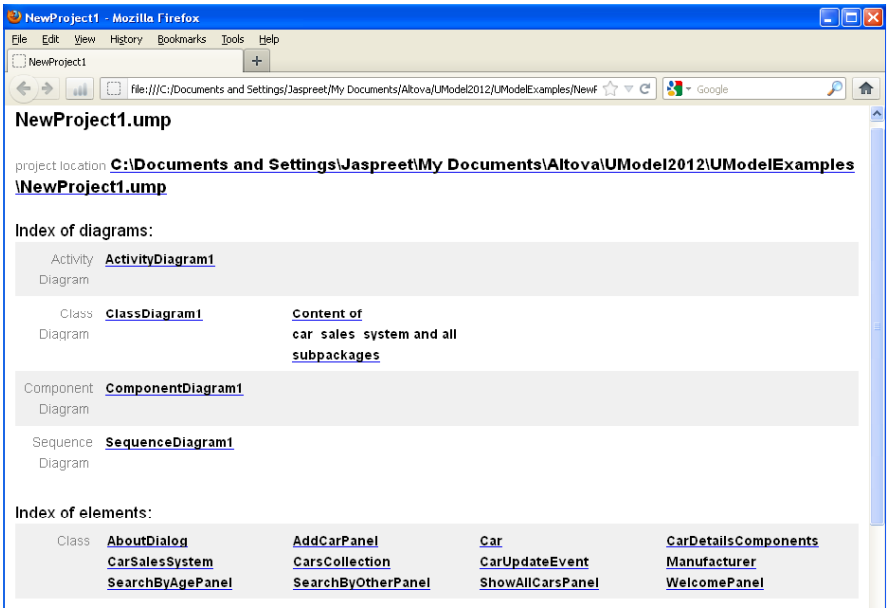

Figure 4.6: Documentation of the care sales management system in Altova UModel Tool.

This is what most programmers mean when using the term software documentation. When creating software, code alone is insufficient. There must be some text along with it to describe various aspects of its intended operation. It is important for the code documents to be thorough, but not so verbose that it becomes difficult to maintain them. Several How-to and overview documentation are found specific to the software application or software product being documented by API Writers. This documentation may be used by developers, testers and also the end customers or clients using this software application.

A document will provide us, our manager and our team with a common vocabulary for talking about the project. A design document can be a powerful tool for a manager because it gives them a view into the project that they don't normally have the technical expertise to see. By listing the benefits we give our manager tangible items that describe why our design is sound.

UModel lets generate sequence diagrams from source code files that have been reverse engineered into UML classes (refer figure 4.7). The resulting sequence diagrams can be an invaluable aid to assist analysis of complex interactions.

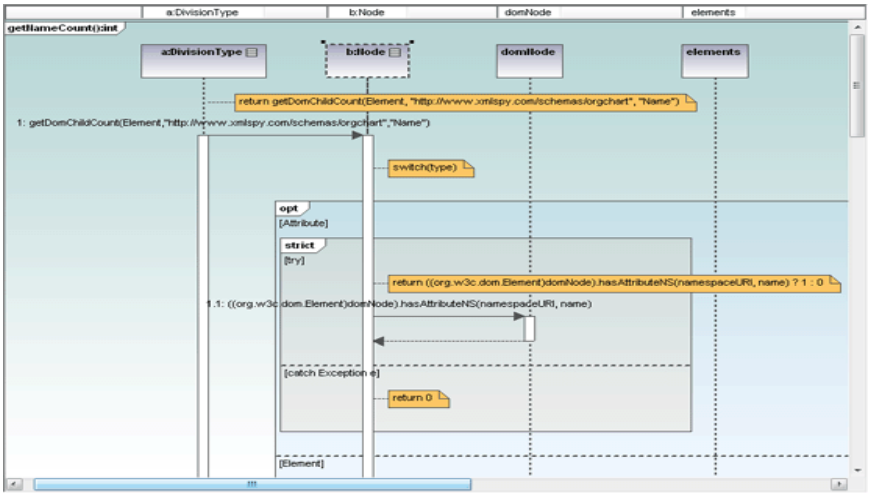

Figure 4.7: Sequence diagram generated from source code in Altova UModel Tool.

The Sequence Diagram Generation dialog offers options to create lists of types and operation names that will not appear in our generated diagram, and to automatically split very large sequence diagrams and hyperlink them for convenient navigation.

d) Identify the Reusable Component using OOAD

To break down the system into components, the software developer needs an understanding of how the object oriented legacy system is built. This task can be accomplished by selecting certain points within the system and expanding the boundaries of those points until all related system elements are included within the boundaries. Here, by using the reverse engineering approach, we are able to extract services points of legacy system and further restructure this system through Object-Oriented Analysis and Design approaches. Object-Oriented Analysis and Design approach is oriented to the tasks and the relevant objects, including their interactions, generalization and composition are used to represent the related terms in domain concept model. Here, we represent the Object-Oriented Analysis and Design approach

from three points of view, i.e. functional, behavioral and structural corresponding to use case, sequence diagram and class diagram. We can extract the structural relationships among the objects from class diagram; the sequence diagram is used to obtain the object usages.

e) Validate and Test Identified Component

Component-based development and software reuse places new demands on software Testing and quality assurance. Testing a software component is basically done to resolve the issues like Checking whether the component meets its specification and fulfill its functional Requirements and Checking whether the correct and complete structural and interaction requirements, Specified before the development of the component, are reflected in the Implemented software system.

Automated tools can be used to test the components like NUnit as a testing framework for all (refer figure 4.8).Net languages. NUnit provides a class library which includes some classes and methods to help us to write test scripts. NUnit provides graphical user interface application to execute all the test scripts and show the result as a success/failure report. NUnit does not create any test scripts by itself but NUnit allows us to use its tools and classes to make the unit testing easier.

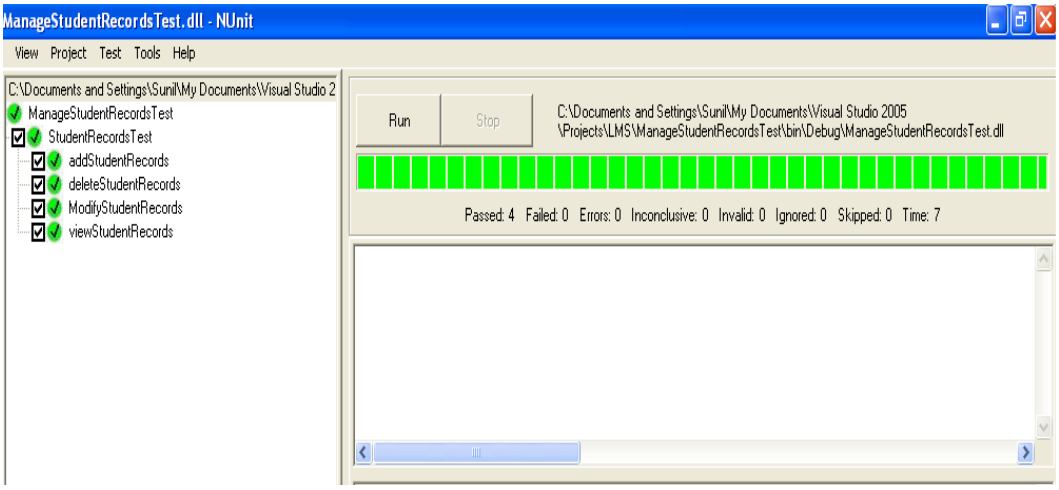

Figure 4.8: Component Testing using NUnit Framework.

f) Store the Components in the Component Repository

After identifying and testing the reusable component, the next task is to store those components with useful information by following a template given below so that components can easily be searched by component searching tool.

Description Templates are the structure of the artifacts, which helps in the storage and retrieval of these components. These templates also help to understand about the artifacts. These templates contain the detail information of the corresponding component. Component Description Template for storing the components is given below.

Name	Description/Tags	Interfaces

g) Component Repository

Organizations will maintain a repository of components from previous projects, external libraries, open-source projects etc. This repository will be continually updated as new classes/systems are developed. Code repositories contain a wealth of valuable information. The repository is effectively a user preference database, a user is a java class and the components employed by a class are items. It is widely believed and empirically proven that component reuse improves both the quality and productivity of software development. Before software components are reused, however, they must be located. Component repository systems provide a means to locate software components. Current component repository systems are designed to support the paradigm of development-with-reuse, which views reuse as a process independent of the whole software development process and rely on programmers to take the reuse initiative. Such systems fall short in supporting programmers who make no attempt to reuse because they do not know the existence of reusable components or they perceive reuse costs more than programming from scratch.

4.1.2 Search, Retrieve, Evaluate and Modify Phase

After the ES phase, the reusable components will be stored in the component repository. As sown in figure 4.9 next job is to search and retrieve the desired component(s) according to the requirement using a component recommendation tool.

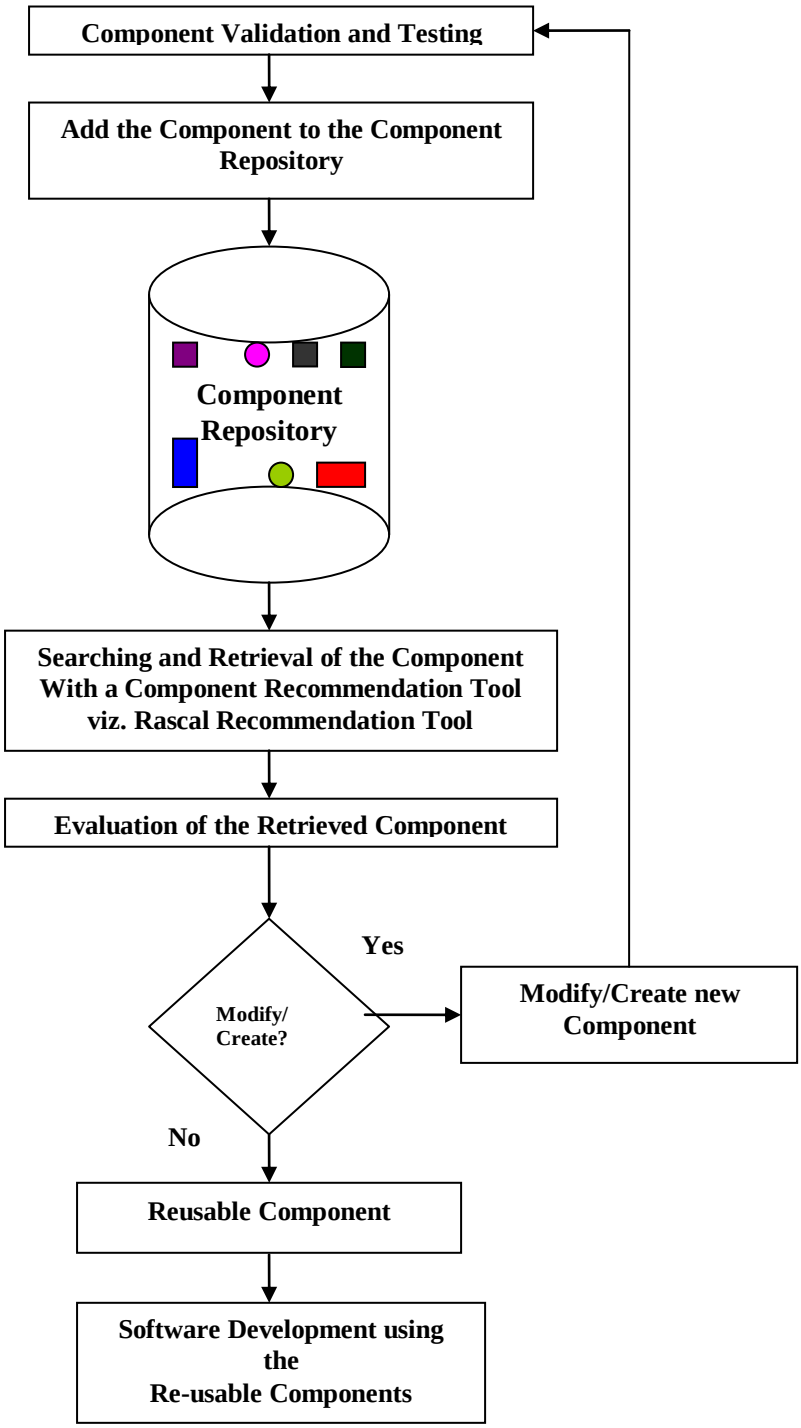

Figure 4.9: SREM phase: Searching and Retrieval of components

Retrieved component is evaluated according to the requirement. If it passes the evaluation process then it'll be used as it is in the software development process otherwise it'll be modified and it'll be again stored in the component repository by passing the component testing process again and it'll also be used in software development process(refer figure 4.9).

a) Search and Retrieve the components

As software organizations mature, their repositories of reusable software components from previous projects will also grow considerably. Remaining conversant with all components in such a repository presents a significant challenge to developers. Indeed the retrieval of a particular component in this large search space may prove problematic. Further to this, the reuse of components developed in an Agile environment is likely to be hampered by the existence of little or no support materials. Searching and Retrieving have long served as the principal techniques for software developers to locate components from component repositories. Searching is direct and fast. Software developers formulate a query, and the repository system returns components that match the query. Formulating queries is a cognitively challenging task because software developers have to overcome the gap from the situational model (i.e., the software developers' understanding of their task) to the system model (i.e., the description of the components in the repository). In Retrieving, software developers determine the usefulness or relevance of the components currently being displayed in terms of their development task, and traverse the associated links in the component repository.

RASCAL, a recommender agent for software components in an agile environment, is developed for two purposes. Firstly we wish to recommend software components that the user is interested in. Secondly, and more importantly, we wish to recommend components that we believe a user actually requires but are unaware of such components existence or the need for such components. RASCAL implicitly gathers information about user usage histories of components and utilizes this information to deduce or infer the need for a particular component or set of components. Specifically the components referred to are Java methods. These inferred methods are then recommended to the user. RASCAL continuously runs in the background, monitoring/updating a user's usage history and frequently makes recommendations**.**

Recommendations are produced using a collaborative filtering approach. RASCAL recommender agent tracks usage histories of a group of developers to recommend to individual developer components that are expected to be needed by that developer. Unlike many traditional recommender systems, we may recommend items that the developer has actually employed previously. Content-based filtering technique for ordering the set of recommended software components and present a comparative analysis of applying this technique to a number of collaborative filtering algorithms. We also investigate the relationship between the number of usage histories collected and recommendation accuracy. Our overall results indicate that RASCAL is a very promising tool for allowing developers discover reusable components at no additional cost [38].

b) Component Evaluation

Component-based development and software reuse places new demands on software Testing and quality assurance. Testing a software component is basically done to resolve the following issues:

• Check whether the component meets its specification and fulfill its functional requirements.

• Check whether the correct and complete structural and interaction requirements, specified before the development of the component, are reflected in the Implemented software system.

c) Modify/Create new Component

After component evaluation the decision will be taken that the retrieved reusable component can be used in the software development as it is or modify/create new component. The modified or newly created component will be first tested and validated then it will be stored in the repository and it will be used in the future project.

Chapter 5
Case Study

5.1 Hostel Management System Case Study

In order to validate the proposed approach, a case study has been performed on Hostel Management System (HMS). The requirement scenario for this case study is taken from the object oriented legacy system developed by Agile Software Development, named Hostel Management System.

5.1.1 Phases of proposed approach for case study:

The steps through which the case study has been done are as follows:

a) Extract and Store phase

i. Analyze of the legacy software system through Reverse Engineering.

ii. Restruct the legacy Software system through OOAD approach.

iii. Apply component identification process.

iv. Validate and Test component.

v. Store the Identified Reusable components in the component Repository.

b) Searching, Retrieving, Evaluating and Modify phase

vi. Search and Retrieve the required component from component repository using a tool.

vii. Use the component in the future projects.

i. Analyze the legacy software system through Reverse Engineering:

By using the reverse engineering tool, named Rational Rose, we analyze the services from the legacy system. It means that the legacy code is used as input to this tool and generating the class diagram through reverse engineering. By using this class diagram as shown in figure 5.1, we are able to restructure our legacy system if needed.

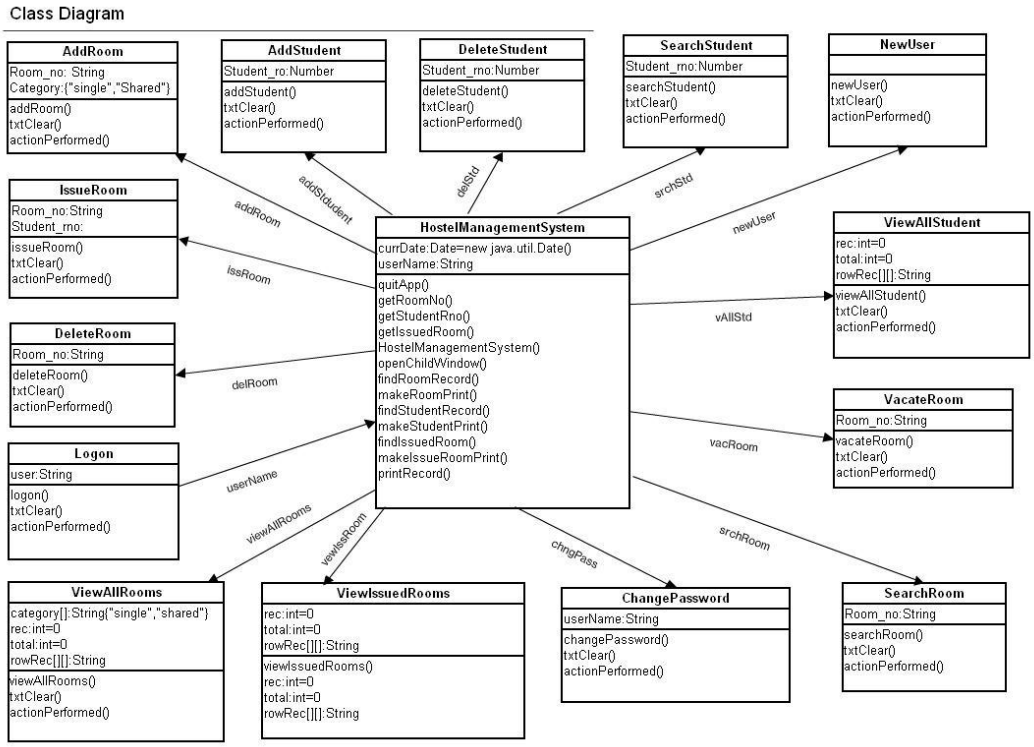

Figure 5.1: Class diagram of legacy system.

The class diagram is the main building block of object oriented modeling. It is used both for general conceptual modeling of the systematic of the application, and for detailed modeling translating the models into programming code. Class diagrams can also be used for data modeling. The classes in a class diagram represent both the main objects and or interactions in the application and the objects to be programmed. In the design of a system, a number of classes are identified and grouped together in a class diagram which helps to determine the static relations between those objects. With detailed modeling, the classes of the conceptual design are often split into a number of subclasses.

ii. Restruct the legacy Software system through OOAD approach:

In this phase, we are considering above class diagram for requirement analysis to build the component based system. Here, we have identified following use cases to find out the functional dependencies, sequence diagram for behavioral relationship and finally design class diagram to predict structural relationships. The identified use cases are: Manage Student Records, Organize Rooms Details, View Room Details, View Student Records, Issue Rooms, and Vacate Room (refer figure 5.2).

iii. Apply component identification process using OOAD

Here, by using the reverse engineering approach, we are able to extract services points of legacy system and further restructure this system through Object-Oriented Analysis and Design approaches. Object-Oriented Analysis and Design approach is oriented to the tasks and the relevant objects, including their interactions, generalization and composition are used to represent the related terms in domain concept model. Here, we represent the Object-Oriented Analysis and Design approach from three points of view, i.e. functional, behavioral and structural corresponding to use case, sequence diagram and class diagram. We can extract the structural relationships among the objects from class diagram; the sequence diagram is used to obtain the object usages.

a) Use case diagrams overview the usage requirements for a system. Use cases provide significantly more value because they describe "the meat" of the actual requirements. Use case diagrams specify the events of a system and their flows. Use case diagram can be imagined as a black box where only the input, output and the function of the black box is known. These diagrams are used at a very high level of design. Then this high level design is refined again and again to get a complete and practical picture of the system.

Although the use cases are not a good candidate for forward and reverse engineering but still they are used in a slight different way to make forward and reverse engineering. And the same is true for reverse engineering. Still use case diagram is used differently to make it a candidate for reverse engineering. (Refer figure 5.2).

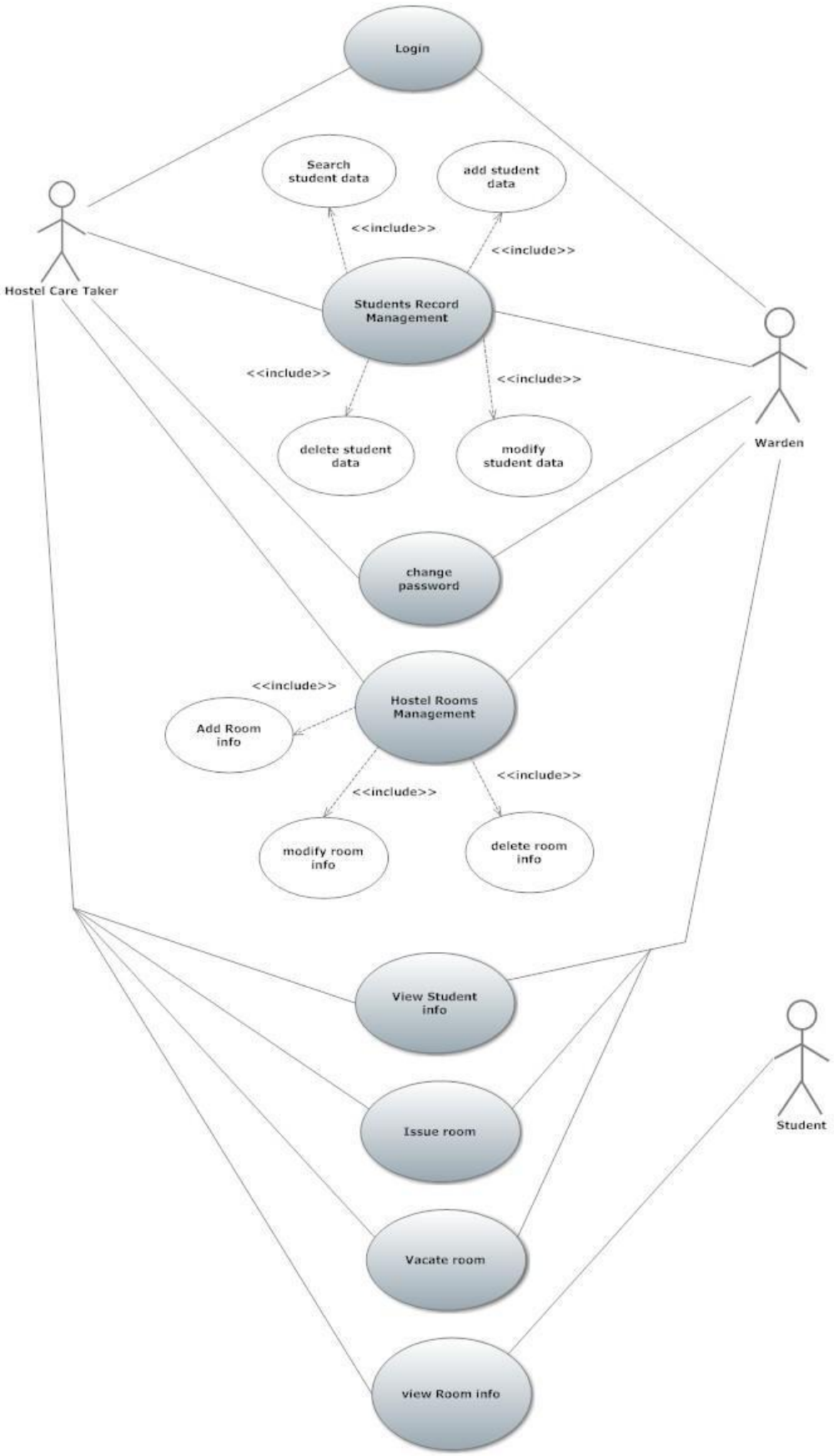

Figure 5.2: Use Case Diagram of HMS.

b) A sequence diagram in a Unified Modeling Language (UML) is a kind of interaction diagram that shows how processes operate with one another and in what order. It is a construct of a Message Sequence Chart (refer figure 5.3 and 5.4). A sequence diagram shows object interactions arranged in time sequence. It depicts the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario. Sequence diagrams typically are associated with use case realizations in the Logical View of the system under development. Sequence diagrams are sometimes called event diagrams, event scenarios, and timing diagrams. A sequence diagram shows, as parallel vertical lines (lifelines), different processes or objects that live simultaneously, and, as horizontal arrows, the messages exchanged between them, in the order in which they occur. This allows the specification of simple runtime scenarios in a graphical manner.

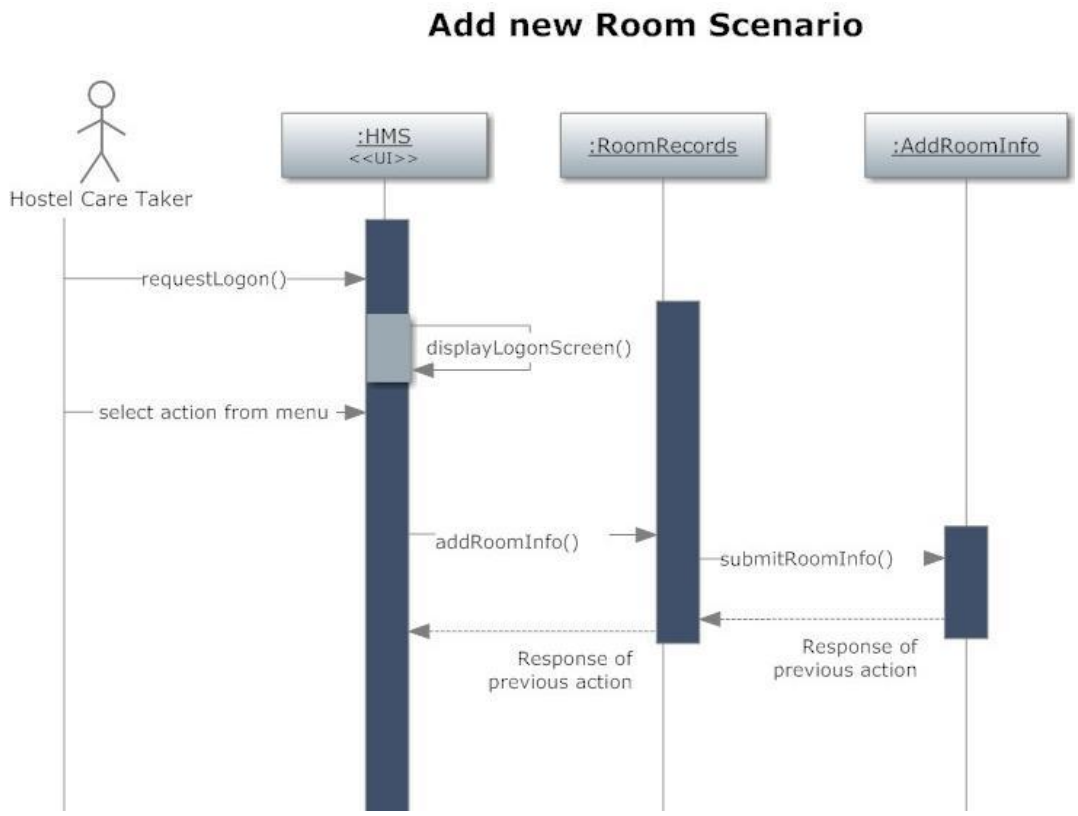

Figure 5.3: Sequence Diagram for Add New Room Scenario.

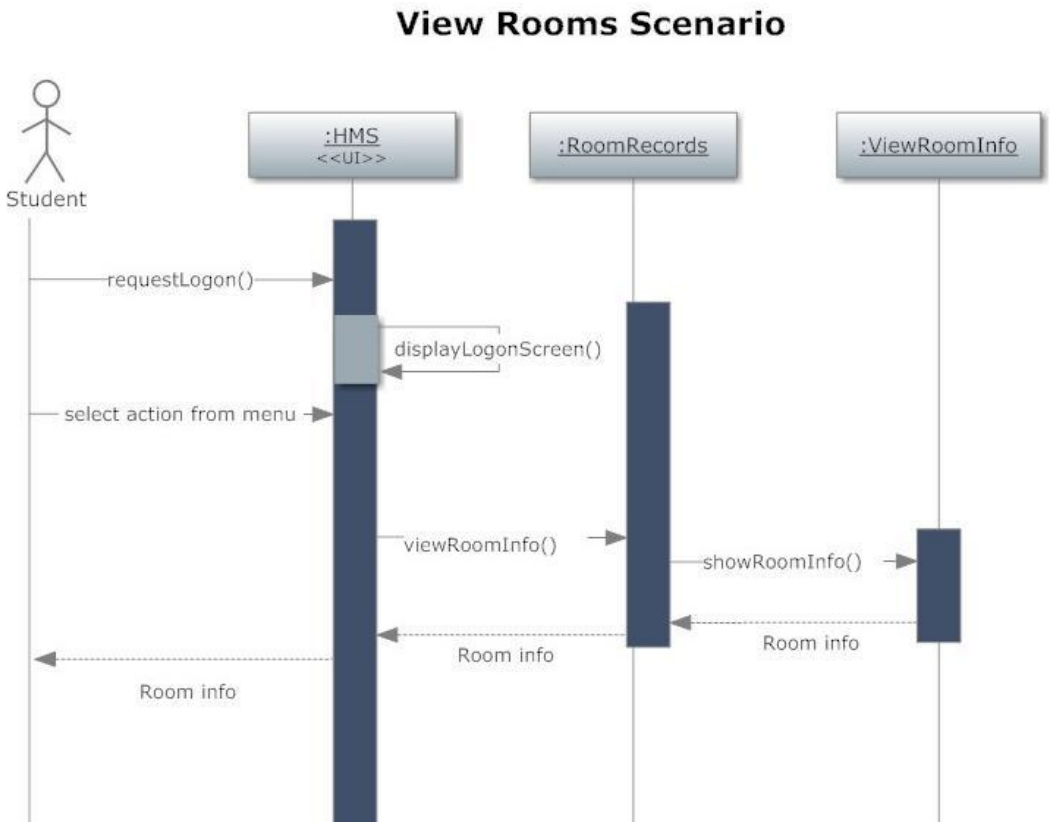

Figure 5.4: Sequence Diagram for View Room Scenario.

One of the primary uses of sequence diagrams is in the transition from requirements expressed as use cases to the next and more formal level of refinement. Use cases are often refined into one or more sequence diagrams. In addition to their use in designing new systems, sequence diagrams can be used to document how objects in an existing (call it "legacy") system currently interact. This documentation is very useful when transitioning a system to another person or organization. It is clear that sequence charts have a number of very powerful advantages. They clearly depict the sequence of events, show when objects are created and destroyed, are excellent at depicting concurrent operations, and are invaluable for hunting down race conditions.

c) UML design class diagrams show software class definitions. They are based on the collaboration diagram. Attribute visibility is shown for permanent connections. Classes are shown with their simple attributes and methods listed. Some attributes are depicted using associations (relationships) rather than actually being listed in the class block. These associated attributes refer to complex objects which should also be shown in the diagram (refer figure 5.5). The collaboration diagram indicates methods to be contained in a class with methods posted as relationships. For instance the Schedule class has a findSeat(route, preference) method.

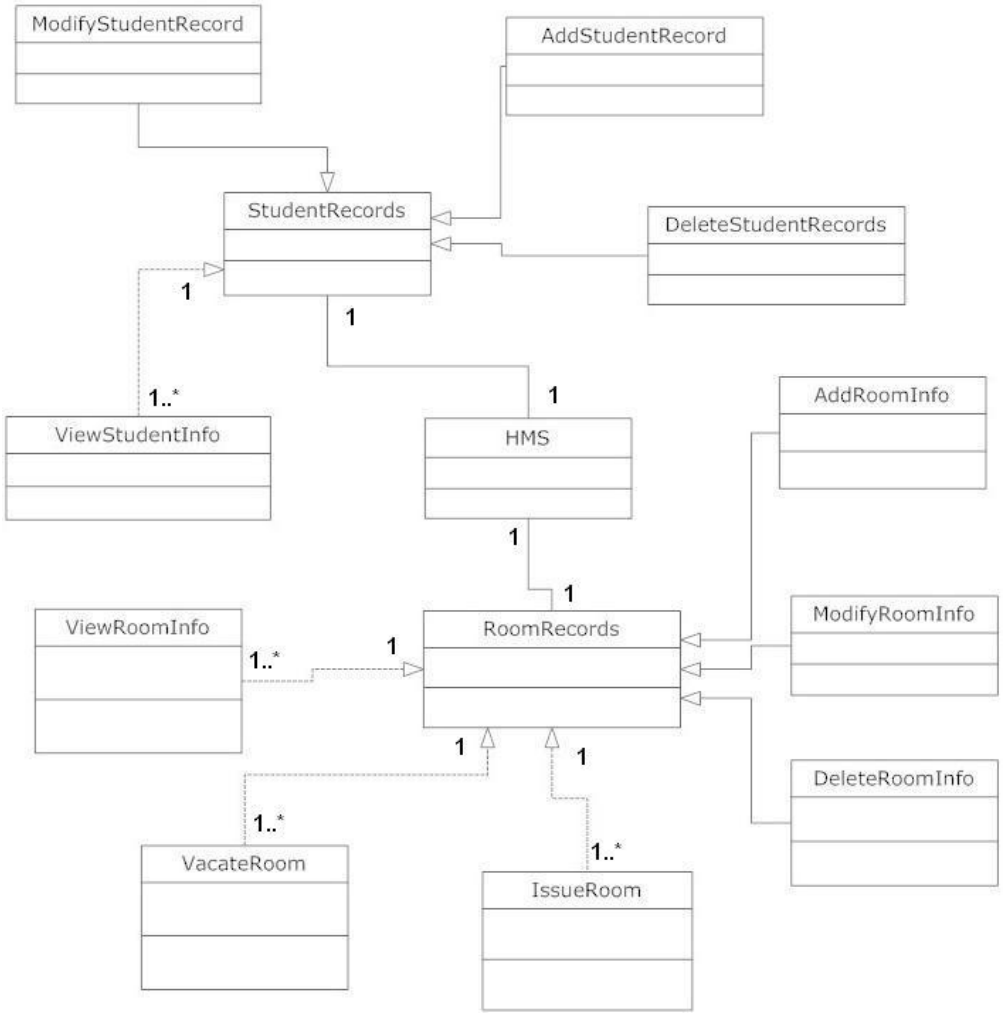

Figure 5.5: Design class diagram.

d) The identified system components and their classes are shown below:

S.No	Reusable component	Classes
1	StudentRecords	AddStudentRecords, DeleteStudentRecords ,ModifyStudentRecords, StudentRecords , ViewStudentRecords
2	RoomDetails	AddRoomDetails, DeleteRoomDetails ,ModifyRoomDetails, RoomDetails, ViewRoomDetails, IssueRooms, VacateRoom

Table 5.1: Reusable components and their classes.

e) Establish Software Component Architecture:

Identified reusable components by using proposed model: StudentRecords, RoomDetails, IssueRooms, VacateRooms, ViewRooms and ViewStudentRecords (refer figure 5.6).

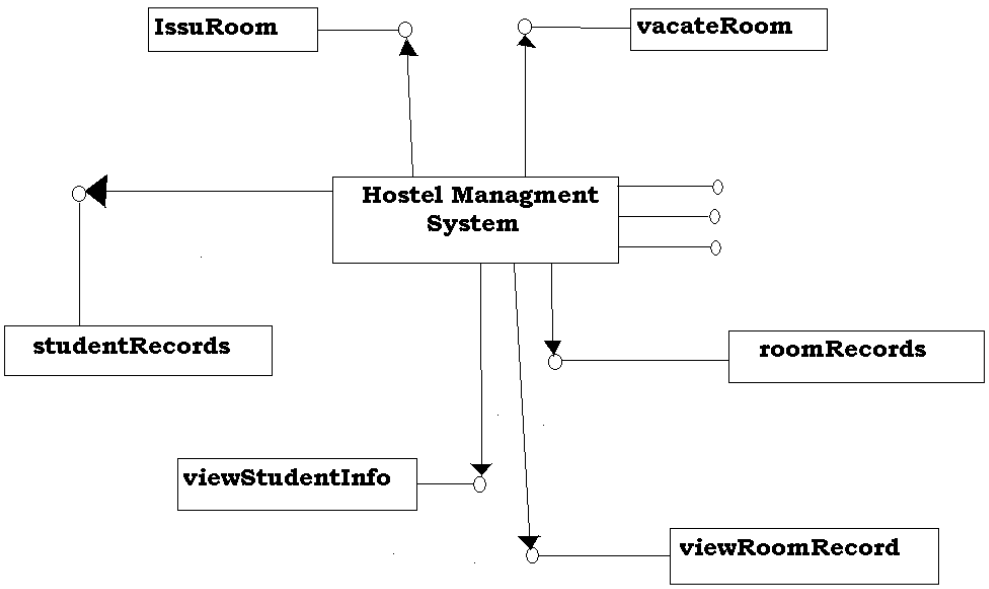

Figure 5.6: Component Architecture of HMS.

iv. Validate and Test Component

Automated tools can be used to test the components like NUnit as a testing framework for all .Net languages (refer figure 5.7). NUnit provides a class library which includes some classes and methods to help us to write test scripts. NUnit provides graphical user interface application to execute all the test scripts and show the result as a success/failure report. NUnit does not create any test scripts by itself but NUnit allows us to use its tools and classes to make the unit testing easier.

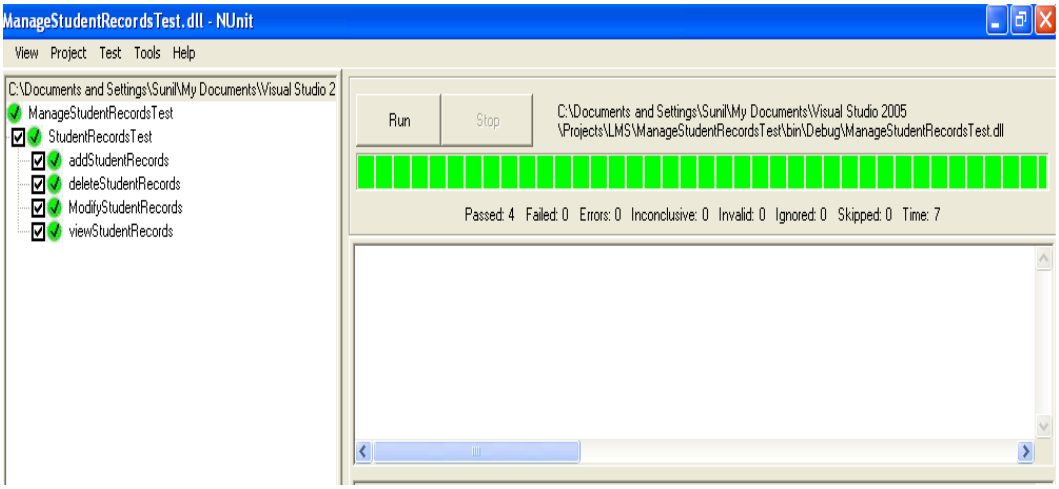

Figure 5.7: Component Testing using Tool.

v. Store the Identified Reusable Components in the Component Repository

After identifying and testing the reusable component, the next task is to store those components with useful information by following a template given below so that components can easily be searched by component searching tool.

Name	Description/Tags	Interfaces

vi. Search and Retrieve the Required Component(s) from Component Repository using a tool

Searching and Retrieving have long served as the principal techniques for software developers to locate components from component repositories. Searching is direct and fast. Software developers formulate a query, and the repository system returns components that match the query. Formulating queries is a cognitively challenging task because software developers have to overcome the gap from the situational model (i.e., the software developers' understanding of their task) to the system model (i.e., the description of the components in the repository). In Retrieving, software developers determine the usefulness or relevance of the components currently being displayed in terms of their development task, and traverse the associated links in the component repository.

vii. Reuse the Component in the future projects

Reuse of components viz. StudentRecords and RoomRecords is illustrated below:

a) Reuse of StudentRecords component

As shown in the figure 5.8 StudentRecords component can be reused in University Management system, Library Management System, Automated Attendance system and other same category of projects where student record keeping and maintenance is one the requirement of the software system.

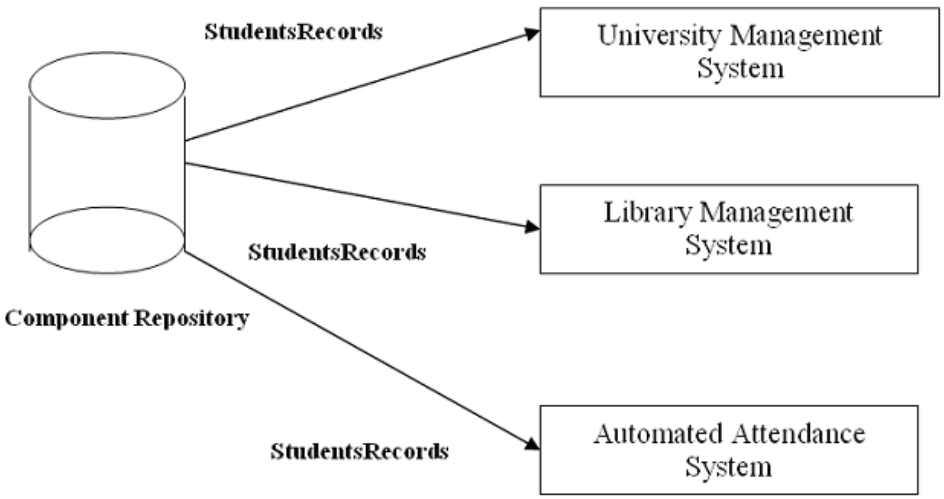

Figure 5.8: Reuse of StudentRecords component in the other projects.

b) Reuse of RoomRecords component

As shown in the figure 5.9 RoomRecords component can be reused in Hotel Management system, University Guest House Management System and other same category of projects where Rooms record keeping and maintenance is one the requirement of the software system.

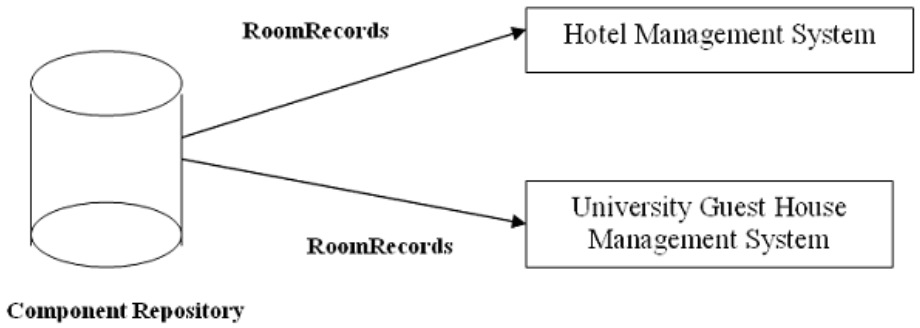

Figure 5.9: Reuse of RoomRecords component in the other projects.

Chapter 6
Conclusion and Future Scope

6.1 Conclusion

Legacy software systems developed with Agile Software Development are often poorly designed and documented, but still perform a good job for the organization's critical application. Due to poor design and documentation it becomes very difficult to reuse the functionality of legacy system in the future projects. The importance of the legacy system cannot be undermined because some of their functions are too valuable to be discarded and too expensive to reproduce. The model based on CASE tools support for extracting the reusable components from the legacy system which are poorly design and documented. It utilizes repository to store and manage the tested components and restructures the new system that finally integrates the new system with the reusable components. The reusability of the software component is most popular way to enhance the productivity and improve the quality and reliability of the new software systems by reducing the development costs. Due to these reasons, it is very important to identify independent components having low coupling and high cohesion. Object oriented Analysis and design approach is used to identify reusable component from object oriented legacy system through models namely use case, interaction diagram and class diagrams has been proposed in this thesis.

Re-engineering based Component Generation and Retrieval model has been validated by using a case-study named Hostel Management System and also its components are tested for reusability and illustrated that how these components can be used in future projects.

6.2 Future Scope

The proposed methodology given in this thesis is not fully automated. Part of the ES and SREM phases are materialized using different CASE tools. But fully automated integrated environment is needed to make the component addition and retrieval more convenient. In future, if Re-engineering based Component Generation and Retrieval

model can be fully automated by an automatic tool then it could be more effective and less time consuming.

References

[1] Joyce, H. & Wendy, W., 2007. “Effective Agile Delivery Toward Globalization”. Available at: www.ibm.com/developerworks/rational/library/oct07/hsieh_wang/.

[2] Joe, L., 2008. “How do I start an Agile Project”. Available at: http://agileconsortium.blogspot.com/2008/0 4/how-do-i-start-agile-project.html.

[3] Jim, H., 2001. “History: The Agile Manifesto”. Available at: http://agilemanifesto.org/history.html.

[4] Jochen, K., 2008. “Agile Portfolio Management”. [E-book] Microsoft Press, Available at Books24x7 http://library.books24x7.com/toc.asp?bokid= 27540.

[5] Jimson, L., 2008. “Plan-Do-Check-Act and the PDCA Deming Cycle”. Available at: http://speedendurance.com/2008/10/14/plan-do-check-act-and-the-pdca-deming-cycle/.

[6] Jim, H., 2000. “Retiring Lifecycle dinosaurs: Using Adaptive Software Development to meet the challenges of a high-speed, high change environment”. Available at: http://www.jimhighsmith.com/articles/Dinosaurs.pdf.

[7] Kathryn, F. & Mats, G., 2008. “Global Software Development and Delivery: Trends and Challenges”. Available at: http://www.ibm.com/developerworks/rational/library/edge/08/jan08/fryer_gothe/index.html.

[8] Michele, S., 2006. “The Agile/ Waterfall Cooperative”. Available at: http://www.rallydev.com/documents/AgileWaterfallCoop-Sliger.pdf.

[9] Kelly, W., 2008. “User Stories. Published on Agile Software Development”. Available at: http://www.agile-software-development.com/2008/01/user-stories.html.

[10] Martin, F., 2006. “Using an Agile Software Process with Offshore Development”. Available at: http://martinfowler.com/article/agileoffshore.html.

[11] Maria, P., Sandra, D. & Casper L., 2008. ”Using SCRUM in a Globally distributed Project: A Case Study”. Available at: http://dl.acm.org/citation.cfm?id=1464416.

[12] Neil, F., 2007. “Global Agile Development: How investing in the Right Team Impacts Long-Term Rewards”. Available at: http://www.agilejournal.com/content/view/436/33/.

[13] Scott W., A., 2006. “The Agile System Development Life Cycle (SDLC)”. Available at:http://www.ambysoft.com/essays/agileLifecycle.html.

[14] Peter, S., 2008. “Prioritizing the Product Backlog”. Available at: http://agilesoftwaredevelopment.com/blog/peterstev/prioritizing-product-backlog.

[15] dtsagile, http://www.dtsagile.com/Agile.

[16] Agile Practices Survey Results: July 2009. Available at:http://www.ambysoft.com/surveys/practices2009.html.

[17] VersionOne Inc., 2008. “3rd Annual urvey: 2008, ‘The State of Agile Development’”. Available at: http://www.versionone.com/pdf/3rdAnnualStateOfAgile_FullDataReport.pdf.

[18] Scott W., A., 2008. “Answering the “Where is the Proof That Agile Methods Work” Question”. Available at:http://www.agilemodeling.com/essays/proof.htm.

[19] SCRUM Alliance, N.D. “What is SCRUM”. Available at:http://www.SCRUMalliance.org/pages/what_is_SCRUM.

[20] Sommerville, I., 2007. “Software Engineering”,8th ed. S.L.: Harlow-Addison-Wesley.

[21] Sanjiv, A., 2007. “Managing Agile Projects”.S.L.: Pearson Education, Inc.VersionOne Inc.,

[22] Jochen, K., 2008. “Agile Portfolio Management”. [E-book] Microsoft Press, Available at: http://library.books24x7.com/toc.asp?bokid= 27540.

[23] Petri, K., 2007. “Extending Software Project Agility with New Product Development Enterprise Agility”. Available at: http://www.interscience.wiley.com.

[24] Vincent, M., 2004. “AODS: Agile Offshore”. Available at: http://codehaus.org/~vmassol/blog/AOSD%20-%20Agile%20Offshore%20-%2020041217.ppt.

[25] Basili, V., Rombach, D., 1991. Support for comprehensive reuse. Department of Computer Science, University of Maryland at College Park, UMIACS-TR-91-23, CSTR- 2606, .Available at: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.133.8831.

[26] M. A. Awed, 2008. A Comparison between Agile and Traditional Software Development Methodologies. Available at: http://pds10.egloos.com/pds/200808/13/85/A_comparision_between_Agile_and_Traditional_SW_development_methodologies.pdf.

[27] M. Griss. , 1996. Systematic Software Reuse: Architecture, Process and Organization are Crucial. Available at: http://martin.griss.com/pubs/fusion1.htm.

[28] V. R. Basili, 1996. et al., "How reuse influences productivity in object-oriented systems," Commun. ACM, vol. 39, Available at: http://portal.acm.org/citation.cfm?id=236184.

[29] R. W. Selby, 1990. "Empirically based analysis of failures in software systems," IEEE Transactions on Reliability, vol. 39, Available at: ieeexplore.ieee.org/iel1/24/2133/00058722.pdf.

[30] S. Henninger, 1997. "An evolutionary approach to constructing effective software reuse repositories," ACM Transactions on Software Engineering and Methodology, vol. 6, Available at: http://dl.acm.org/citation.cfm?id=248242.

[31] A. Tomer, et al., 2004. "Evaluating software reuse alternatives: A model and its application to an industrial case study," IEEE Transactions on Software Engineering, vol. 30. Available at: http://www.cs.technion.ac.il/~tomera/publications/Cost%20of%20Reuse/tomer-et-al-TSE_2004.pdf.

[32] W. B. Frakes and K. Kang, 2005. "Software reuse research: Status and future," IEEE Transactions on Software Engineering, vol. 31. Available at: http://v1.arnetminer.org/viewpub.do?pid=1128116.

[33] C. W. Krueger, 1992. "Software reuse," ACM Computing surveys, vol. 24. Available at: http://dl.acm.org/citation.cfm?id=130856.

[34] H. Mili, et al., 1995. "Reusing software: issues and research directions," Software Engineering, IEEE Transactions on, vol. 21. Available at: www.cin.ufpe.br/~in1045/papers/art02.pdf.

[35] Rumbaugh James et.al. , 1991. Object oriented modeling and design.Publisher: Prentice Hall - Computers - 500 pages.

[36] Chikofsky E.J., Cross II J.H., 1990. Reverse Engineering and Design Recovery: a Taxonomy, IEEE Software, volume 7. Available at: http://www.cs.cmu.edu/~aldrich/courses/654-sp05/ReengineeringTaxonomy.pdf.

[37] Ian Sommerville, 2004. Software Engineering, 7th edition. Chapter 18. Available at: http://www.comp.lancs.ac.uk/computing/resources/IanS/SE7/Presentations/PDF/ch18.pdf.

[38] Frank McCarey, 2005. Rascal: Agile Software Reuse Recommender – IEEE. Available at: http://ieeexplore.ieee.org/iel5/11128/35643/01690844.pdf.

[39] Asit Kumar Gahalaut, 2010. REVERSE ENGINEERING: AN ESSENCE FOR SOFTWARE RE-ENGINEERING AND PROGRAM ANALYSIS. Available at: http://www.ijest.info/docs/IJEST10-02-06-131.pdf.

[40] Object Management Group (OMG), http://www.omg.org/mda.

List of Publications

1. Jaspreet Singh, Ashima Singh, 2012, "Agile Software Development and Reusability", in Proceedings International Conference on Competitiveness & Innovativeness in Engineering, Management & Information Technology (ICCIEMI-2012), 29th January, 2012.

2. Jaspreet Singh, Ashima Singh, 2012, "Agile Software Development and Reusability", International Journal of Research in Engineering and Applied Sciences, ISSN: 2249-3905, Volume 2, Issue 2, pp. 1182-1188, February 2012.