Software Artifacts based Change Controller

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

Master of Engineering

in

Software Engineering

Thapar University, Patiala

By:

Tarun Kumar Agrawal

(80631023)

Under the supervision of

Mr. Rajesh Kumar Bhatia

Assistant Professor

JUNE 2008

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

THAPAR UNIVERSITY

PATIALA – 147004

Certificate

I hereby certify that the work which is being presented in the thesis entitled, **"Software Artifacts based Change Controller"**, 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 *Mr. Rajesh Kumar Bhatia* and refers other researcher's works which are duly listed in the reference section.

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

(Tarun Kumar Agrawal)
Roll No. 80631023

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

(Mr. Rajesh Kumar Bhatia)
Supervisor
Computer Science and Engineering Department
Thapar University
Patiala

Countersigned by

(Dr. Seema Bawa)
Professor & Head
Computer Science & Engineering. Department
Thapar University
Patiala

(Dr. R. K. Sharma)
Dean(Academic Affaris)
Thapar University,
Patiala.

Acknowledgement

At first, I would like to thank Computer Science & Engineering Department which allowed me to do such type of research work. No volume of words is enough to express my gratitude towards my guide, Mr. Rajesh Kumar Bhatia, Thapar University, Patiala, who has been very concerned and has aided for all the material essential for the preparation of this thesis report. They have helped me to explore this vast topic in an organized manner and provided me with all the ideas on how to work towards a research-oriented venture.

I am also thankful to Dr. Seema Bawa, Head of Department, and Computer Science & Engineering Department for the motivation and inspiration that triggered me for the thesis work.

I would also like to thank the staff members and my colleagues who were always there at the need of the hour and provided with all the help and facilities, which I required, for the completion of my thesis work.

Most importantly, I would like to thank my parents and the almighty for showing me the right direction out of the blue, to help me stay calm in the oddest of the times and keep moving even at times when there was no hope.

Tarun Kumr Agrawal

80631023

Abstract

Change is the law of nature. Software industry is not also an exception. Today software industry is facing a problem of changes in the software due to adaptation, versioning, variants, maintenance, bugs. Changes are to be made in software due to various reasons. The reasons include change of requirements, missing requirements, change of design, errors in coding, modification in coding, new market conditions, business growth/downsizing, budgetary or scheduling constraints. Modules of software may be low or high coupled. Modules of a software may also be low or high cohesive.

Whenever one error occurs in one module, it can lead errors in same module as well as in other modules. As a change is made in some artifact of the software, the change will induce errors in other software artifacts. Thus a change in an artifact induces side-effects in the software. Software developers have to put a lot of effort to find out the affected software artifacts due to a change in one artifact. Thus the development cost of software and time to develop software increases.

There are a number of tools available that find the affect of change in one code module on other code modules. The proposed method not only finds the affected code module but also other software artifacts due to change in any software artifact .The proposed method stores the software artifacts in the software repository and associates the software artifacts with each other. Whenever a change occurs in a software artifact, the proposed method finds the affected software artifacts in an efficient way. Thus the software development effort and cost reduces.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Change is the mother of invention. As the time passes, the technology changes, human beings wants more facilities. As the era of software is maturing, the requirements of user are changing according to time. The old things become obsolete. The new things come into existence. Human being wants more benefits from the software. So changes are to be performed in the software. These changes are due to errors in software or the changing requirements of the user. Changes are to be performed in the software artifacts accordingly. So change management and configuration management is necessary in the software.

1.1 Software Artifact

An artifact is one of many kinds of tangible byproduct produced during the development of software. Some artifacts for e.g., use cases, class diagrams, and other UML models, requirements and design documents help in describing the function, architecture, and design of software. Other artifacts are concerned with the process of development itself, such as project plans, business cases, and risk assessments.

Build tools often refer to source code compiled for testing as an artifact, because the executable is necessary to carrying out the testing plan. Without the executable to test, the testing plan artifact is limited to non-execution based testing. In non-execution based testing, the artifacts are the walkthroughs, inspections and correctness proofs. Walkthrough is a step-by-step review of a specification, usability features or design before it is handed off to the technical team for development. Inspection is used throughout the software development life cycle to assess and improve the quality of various work products for ex. Items to be inspected in a design inspection might include completeness of the design corresponding to user requirements, correctness of the interface between modules. On the other hand, execution based testing requires at minimum two artifacts: a test suite and the executable. An artifact occasionally may be used to refer to the released code in the case of a code library or released executable in

the case of a program produced but the more common usage is in referring to the byproducts of software development rather than the product itself. Software artifacts are given as following:

Executable Code: The essence of a piece of the executable code is the function that it computes. Executable code is typically represented in machine-readable form.

Source Code: Source code is represented by programming languages, and can be indexed by means of its structural properties, as well as its functional properties. Executable code and source code are referred to as software components or simply as components.

Requirement: Whereas code artifacts are executable, requirements are not.

Design: Design are the generic representation of design decisions, their essence is the design/problem-solving knowledge that they capture. In contrast to code, designs are not executable: in contrast to specification assets, they capture structural information rather then functional information. They are represented by architecture that can be instantiated in different ways to produce concrete designs.

Text Data: Consider a system under software development. Integration-testing of this system has been performed using some text data. the same data may be reused to test a similar product, which has a similar set of inputs but different output conditions for example that the same text data can be used to text a procedure that converts decimal to binary number , and a procedure that converts decimal to octal number .

Documentation: natural-Language documentation that accompanies a reusable artifact can conceivably be considered as a reusable asset itself; in addition, many reusable assets like specification and design can be represented as natural-Language documentation.

Architecture: Software architecture defines the set of a software systems the aggregate of a set of components that exchange data. Architectures are represented by means of specialized notations and are indexed by means of their architectural features. The main emphasis is given on the following software artifacts:

- Requirement
- Design
- Code
- Test

1.1.1 Requirement

In principle, requirements should state what the system should do but not how the system does. It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification. This is inevitable as requirements may serve a dual function

- May be the basis for a bid for a contract , therefore must be open to interpretation
- May be the basis for the contract itself, therefore must be defined in detail.

1.1.1.1 Types of Requirement

- User requirements
- System requirements

User Requirements

User requirements are written for customers. User requirements should describe functional and non-functional requirements in such a way that they are understandable by system users who don't have detailed technical knowledge. User requirements are defined using natural language, tables and diagrams as these can be understood by all users. User requirements are of following types

a) Functional requirements
b) Non-functional requirements
c) Domain requirements

a) Functional Requirements

Functional requirements are the statements of services the system should provide how the system should react to particular inputs and how the system should behave in particular situations Functional requirements describe functionality or system services. These requirements depend on the type of software, expected users and the type of system where the software is used

b) Non-functional Requirements:

These requirements impose constraints on the services or functions offered by the system such as timing constraints. These requirements also specify constraints on the development process, standards, etc.

c) Domain Requirements

Requirements that come from the application domain of the system and that reflect characteristics of that domain

System Requirements

System requirements set out detailed descriptions of the system's functions, services and operational constraints.

1.1.1.2 Requirements Completeness and Consistency

In principle, requirements should be both complete and consistent.

Complete: Requirements should include descriptions of all facilities required.

Consistent: There should be no conflicts or contradictions in the descriptions of the system facilities.

1.1.1.3 Requirements Elicitation

Requirements elicitation is the process of discovering the requirements for a system by communication with customers, system users and others who have a stake in the system development [17].

Requirements Elicitation Techniques

- Interviewing and questionnaires
- Braining Storming
- Storyboarding
- Use Cases
- Role Playing
- Prototyping

Interviewing and Questionnaires

- Simple direct technique
- Context-free questions can help achieve bias-free interviews
- Then, it may be appropriate to search for undiscovered requirements by exploring solutions.
- Convergence on some common needs will initiate a "requirements repository" for use during the project.

- A questionnaire is no substitute for an interview.
- Goal is to prevent prejudicing the user's response to the questions.

Examples:

Who is the user?

Who is the customer?

Are their needs different?

Where else can a solution to this problem be found?

Brainstorming

- Brainstorming involves both idea generation and idea reduction.
- The most creative, innovative ideas often result from combining, seemingly unrelated ideas.
- Various voting techniques may be used to prioritize the ideas created.
- Although live brainstorming is preferred, web-based brainstorming may be a viable alternative in some situations

Rules for Brainstorming

- Do not allow criticism or debate.
- Generate as many ideas as possible
- Mutate and combine ideas
- Idea Reduction
- Pruning ideas that are not worthy of further discussion
- Grouping of similar ideas into one super topic
- Prioritize the remaining ideas

Storyboarding

- Storyboards can be positive, active, or inactive.
- Storyboards identify the players, explain what happens to them, and describes how it happens.
- Make the storyboard sketchy, easy to modify.
- Storyboard early and often on every project with new or innovative content.

Use Cases

- Use Cases, like storyboards, identify who, what, and how of system behavior.

- Use Cases describe the interactions between a user and a system, focusing on what they system "does" for the user.
- The Use Case model describes the totality of the systems functional behavior.

Role Playing – variant on use cases

- Role playing allows stakeholders to experience the user's world from the user's perspective.
- A scripted walkthrough may replace role playing in some situations, with the script becoming a live storyboard.

Prototyping

- Prototyping is especially effective in addressing the "Yes, But" and the "Undiscovered Ruins" syndromes.
- A software requirements prototype is a partial implementation of a software system, built to help developers, users, and customers better understand system requirements.
- Prototype the "fuzzy" requirements: those that, although known or implied, are poorly defined and poorly understood.

1.1.2 Design

Software is designed according to software requirements. UML language is used for designing the software. The UML is a mostly graphical modeling language that is used to express designs. The UML (Unified/Universal Modeling Language) is a standardized visual specification language for object modeling [5]. UML is a general-purpose modeling language that includes a graphical notation used to create an abstract model of a system, referred to as a UML model. UML is defined as a standard language for specifying, constructing and documenting all sorts of systems, software or other types. UML was initially developed by James Rumbaugh, Grady Booch and Ivar Jacobson in order to give all developers a standard communication language in which they could discuss the general architecture of the program. UML does not have any dependencies with respect to any technologies or languages. This implies that UML can be used to model applications and systems based on either of the current hot technologies; for example, J2EE and .NET.

UML is made up of nine diagrams that can be used to model a system at different points of time in the software life cycle of a system. The nine UML diagrams are:

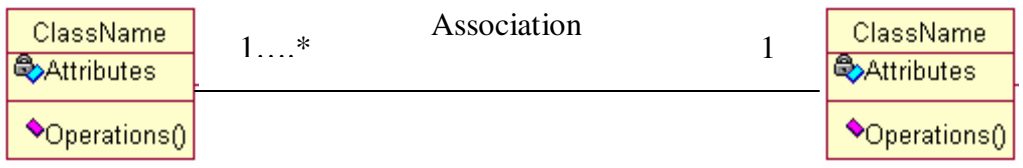
- **Use Case Diagram:** The use case diagram is used to identify the primary elements and processes that form the system. The primary elements are termed as actors and the processes are called use cases. The use case diagram shows which actors interact with each use case.

Figure1.1: Use Case symbols

- **Class Diagram:** The class diagram is used to refine the use case diagram and define a detailed design of the system. The class diagram classifies the actors defined in the use case diagram into a set of interrelated classes. A class contains attributes and operations. An attribute is a property of the class. For ex. A student has id, name and age. The attributes of the class student will be id, name and age. Each class may have certain attributes that uniquely identify the class. Each class in the class diagram may be capable of providing certain functionalities. These functionalities provided by the class are termed as methods of the class. The relationship between two classes is following:

  Association

  An association is a relationship between two classes.



Class Employee Class Manager

Figure 1.2: Association between two classes

In the figure 1.2, one or more employees may be associated with one manager.

Dependency

A dependency is a semantic relationship between two things in which a change to one thing (the independent thing) may affect the semantics of other thing (the dependent thing)

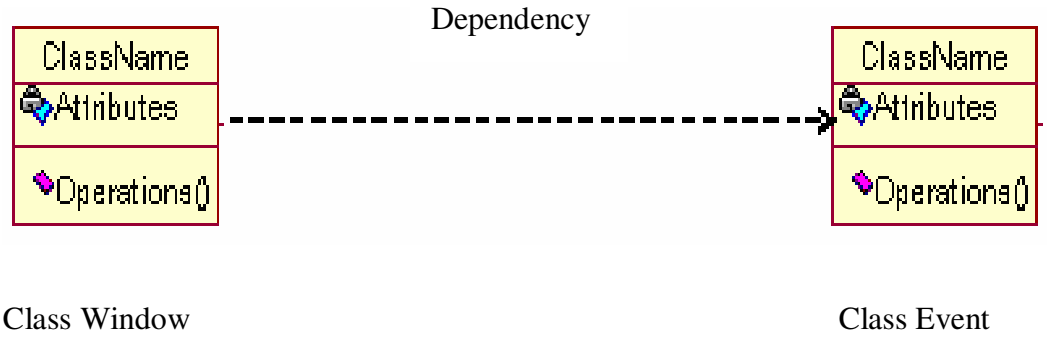


Figure 1.3: Dependency between two classes

In figure 1.3 class Window depends on Event.

Generalization

A generalization is a generalization/specialization relationship in which objects of the specialized class (the child) inherit attributes and operations from the objects of the generalized class(the parent).

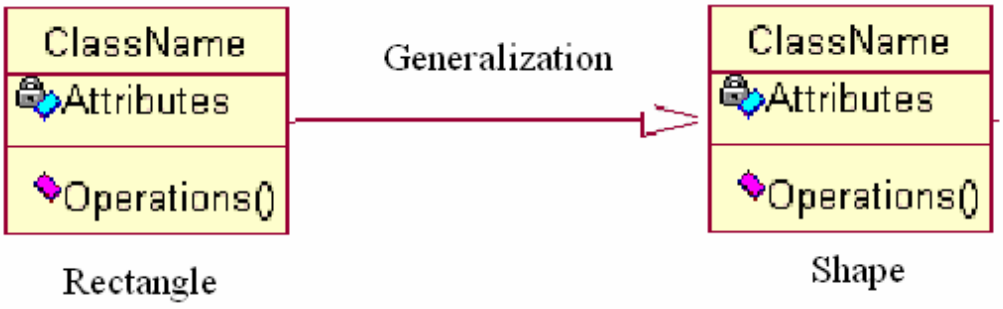


Figure 1.4: Generalization between two classes

In figure 1.4 class Rectangle is derived from parent class Shape

Aggregation

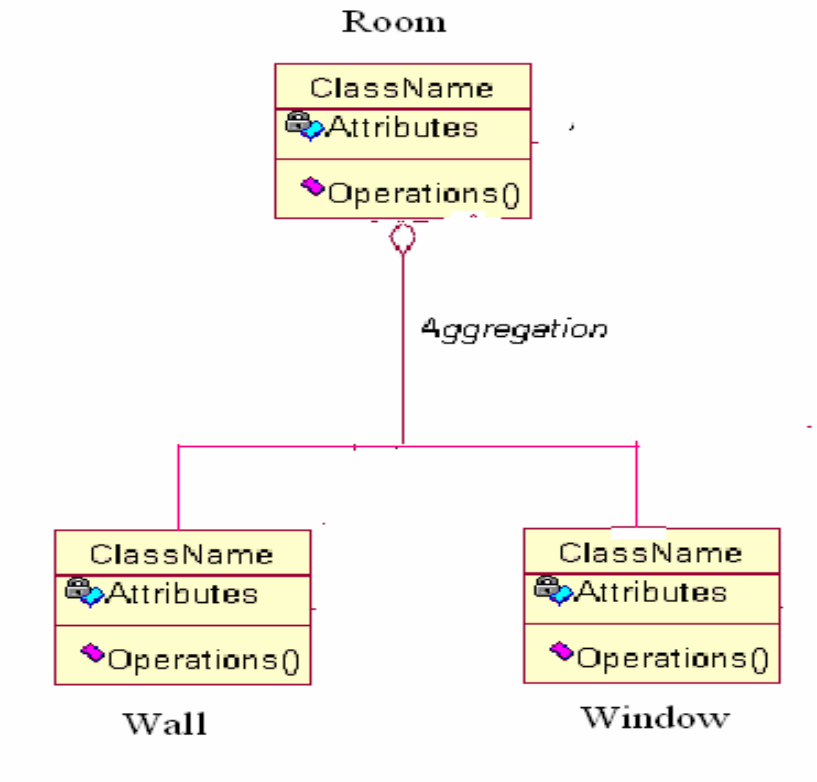


Figure 1.5: Aggregation of classes

In figure 1.5 class Room is an aggregate of the classes Wall and Window.

- **Object Diagram:** The object diagram is a special kind of class diagram. An object is an instance of a class. This essentially means that an object represents the state of a class at a given point of time while the system is running. The object diagram captures the state of different classes in the system and their relationships or associations at a given point of time.
- **State Diagram:** A state diagram, as the name suggests, represents the different states that objects in the system undergo during their life cycle. Objects in the system change states in response to events. In addition to this, a state diagram also captures the transition of the object's state from an initial state to a final state in response to events affecting the system. State transition system is shown in figure 1.6.

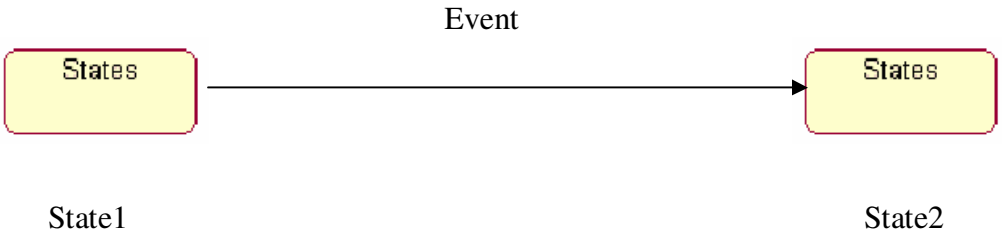


Figure 1.6: State transition diagram

- **Activity Diagram:** The process flows in the system are captured in the activity diagram. Similar to a state diagram, an activity diagram also consists of activities, actions, transitions, initial and final states, and guard conditions.
- **Sequence Diagram:** A sequence diagram represents the interaction between different objects in the system. The important aspect of a sequence diagram is that it is time-ordered. This means that the exact sequence of the interactions between the objects is represented step by step. Different objects in the sequence diagram interact with each other by passing messages.
- **Collaboration Diagram:** A collaboration diagram groups together the interactions between different objects. The interactions are listed as numbered interactions that help to trace the sequence of the interactions. The collaboration diagram helps to identify all the possible interactions that each object has with other objects.

Figure 1.7: Collaboration diagram

- **Component Diagram:** The component diagram represents the high-level parts that make up the system. This diagram depicts, at a high level, what components form part of the system and how they are interrelated. A component diagram depicts the components culled after the system has undergone the development or construction phase.

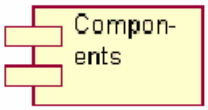


Figure 1.8: Component

- **Deployment Diagram:** The deployment diagram captures the configuration of the runtime elements of the application. This diagram is by far most useful when a system is built and ready to be deployed.

1.1.3 Code

Code is the implementation of design. Code is in the form of class. A class A is said to be dependent on other class B if class A inherits its attributes and functions from class B. Requirements can be modeled into design by using Rational rose. Class diagrams are created during design activity. In the design activity, class diagrams are produced. during coding, classes are created from class diagram.

1.1.4 Test

When the implementation of software is over, it's very essential to test the software. Testing is a process to execute the program with the intent of finding errors.

1.1.4.1 Testing Principles

Testing principles are applied for designing test cases. Testing principles are following:

- All tests should be traceable to customer requirements.
- Tests should be planned before testing begins.
- Testing should begin "in the small" and progress towards testing "in the large"
- Testing should be conducted by independent third party.
- The Pareto principle applies to software testing i.e. 80 percent of all errors uncovered during testing will likely be traceable 20 percent of all program components.

1.1.4.2 Attributes of a Good test

- A good test has a high probability of finding an error.

- A good test is not redundant
- A good test should be neither too simple nor too complex.
- A good test should be best of breed. If a group of tests have a similar intent, time and resource limitations then only a subset of these tests must be executed. This subset of tests has the highest likelihood of uncovering a whole class of errors.

1.1.4.3 Test -Case Design

Software can be tested in following two ways:

- Black-box testing
- White-box testing

Black-Box Testing

The tester views the program or the unit to be tested as black box whose internals are not known. The test cases are solely derived from the specification but not from information about the implementation [23].

Black box tests are used to demonstrate that software functions are operational i.e. input is properly accepted and output is correctly produced black-box testing is conducted at the software interface.

White-Box testing

Logical paths through the software are tested by providing test cases that exercise specifics set of conditions and/or loops. Software should be tested through a limited number of logical paths. Test cases are derived from the internal structure of the unit or the whole program and the specification to compare the actual vs. the expected results.

1.2 Software Maintenance

Software maintenance is the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.

1.2.1 Need for Maintenance

Maintenance is needed to ensure that the software continues to satisfy user requirements. Maintenance inapplicable to software developed using any software lifecycle model for

example, spiral. The system changes due to corrective and non-corrective software actions. Maintenance must be performed in order to:

- Correct faults
- Improve the design
- Implement enhancements
- Interface with other systems
- Adapt programs so that different hardware, software, system features, and telecommunications facilities cane used
- Migrate legacy software
- Retire software

The maintainer's activities comprise four key characteristics

- Maintaining control over the software's day-to-day functions
- Maintaining control over software modification
- Perfecting existing functions
- Preventing software performance from degrading to unacceptable levels

1.2.2 Types of Software Maintenance

- Corrective maintenance
- Adaptive maintenance
- Perfective maintenance
- Preventive maintenance

Corrective Maintenance

Corrective maintenance is reactive modification of a software product performed after delivery to correct discovered problems. It deals with the repair of faults or defects found. A defect can result from design errors, logic errors and coding errors.

Adaptive Maintenance

Adaptive Modification is modification of a software product performed after delivery to keep a software product usable in a changed or changing environment. Adaptive maintenance consists of adapting software to changes in the environment, such as the hardware or the operating system.

Perfective Maintenance

Perfective maintenance is modification of a software product after delivery to improve performance or maintainability Perfective maintenance mainly deals with accommodating to new or changed user requirements

Preventive Maintenance:

Modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults. It concerns activities aimed at increasing the system's maintainability, such as updating documentation, adding comments, and improving the modular structure of the system

1.2.3 Key issue in Software maintenance

Impact Analysis

Impact analysis describes how to conduct, cost effectively, a complete analysis of the impact of a change in existing software. Maintainers must possess an intimate knowledge of the software's structure and content. They use that knowledge to perform impact analysis, which identifies all systems and software products affected by a software change request and develops an estimate of the resources needed to accomplish the change. Additionally, the risk of making the change is determined.

The objectives of impact analysis are:

- Determination of the scope of a change in order to plan and implement work
- Development of accurate estimates of resources needed to perform the work
- Analysis of the cost/benefits of the requested change
- Communication to others of the complexity of a given change

1.3 Software Configuration Management

Software configuration management is a set of management disciplines within the software engineering process to develop a baseline [7]. Software configuration management encompasses the disciplines and techniques of initiating, evaluating and controlling change to software products during and after the software engineering process. New versions of software systems are created as they change:

- For different machines/OS;

- Offering different functionality
- Tailored for particular user requirements.

CM (Configuration management) is concerned with managing evolving software systems. It focuses on following principles.

- System change is a team activity;
- CM aims to control the costs and effort involved in making changes to a system.
- Involves the development and application of procedures and standards to manage an evolving software product.
- CM may be seen as part of a more general quality management process.
- When released to CM, software systems are sometimes called *baselines* as they are a starting point for further development.

1.3.1 CM Standards

- CM should always be based on a set of standards which are applied within an organization.
- Standards should define how items are identified; how changes are controlled and how new versions are managed.
- Standards may be based on external CM standards (e.g. IEEE standard for CM).

1.3.2 Managing Software Configurations

Software configuration management is a project function with the goal to make technical and managerial activities more effective. Software configuration management can be staffed in several ways:

- A single team performs all software configuration management activities for the whole organization
- A separate configuration management team is set up for each project
- All the software configuration management activities are performed by the developers themselves
- Mixture of all of the above

1.3.3 Configuration Management Planning

- All products of the software process may have to be managed such as specifications, designs, programs, test data, user manuals.
- Thousands of separate documents may be generated for a large, complex software system.

1.3.4 The CM Plan

- Defines the types of documents to be managed and a document naming scheme.
- Defines who takes responsibility for the CM procedures and creation of baselines.
- Defines policies for change control and version management.
- Defines the CM records which must be maintained.
- Describes the tools which should be used to assist the CM process and any limitations on their use.
- Defines the process of tool use.
- Defines the CM database used to record configuration information.
- May include information such as the CM of external software, process auditing, etc.

1.3.5 Configuration Management Activities

Configuration Item Identification: modeling of the system as a set of evolving components

Promotion Management: the creation of versions for other developers

Release Management: the creation of versions for the clients and users

Change Management: the handling, approval and tracking of change requests

Branch Management: the management of concurrent development efforts

Variant Management: the management of versions intended to coexist

Configuration Item

A configuration item an aggregation of hardware, software, or both, that is designated for configuration management and treated as a single entity in the configuration management process. Software configuration items are not only program code segments but all type of documents according to development, for ex.

- all type of code files
- drivers for tests
- analysis or design documents
- user or developer manuals
- system configurations e.g. version of compiler used

Baseline

A baseline is a specification or product that has been formally reviewed and agreed to by responsible management that thereafter serves as the basis for further development, and can be changed only through formal change control procedures

Many naming scheme for baselines exist (1.0, 3.14159, 6.01a.). A 3 digit scheme is quite common. For ex.: Mac OS X 10.3.6(Release, version, revision) A revision is a new version of an item that is intended to replace the old version of the item

Configuration Item Identification

- Large projects typically produce thousands of documents which must be uniquely identified.
- Some of these documents must be maintained for the lifetime of the software.
- Document naming scheme should be defined in such a way so that related documents may be easily identified.

A hierarchical scheme with multi-level names is probably the most flexible approach.

For ex. PCL-TOOLS/EDIT/FORMS/DISPLAY/AST-INTERFACE/CODE

The Configuration Database

- All CM information should be maintained in a configuration database.
- This should allow queries about configurations to be answered:
  1. Who has a particular system version?
  2. What platform is required for a particular version?
  3. What versions are affected by a change to component X?
- How many reported faults in version T?
- The CM database should preferably be linked to the software being managed.

1.3.6 Change Management

Software systems are subject to continual change requests from users, developers and market forces. Change management is concerned with keeping track of these changes and ensuring that they are implemented in the most cost-effective way. Change management is the handling of change requests. A change request leads to the creation of a new release

1.3.7 Version and Release Management

- Invent an identification scheme for system versions.
- Plan when a new system version is to be produced.
- Ensure that version management procedures and tools are properly applied.
- Plan and distribute new system releases

1.3.7.1 Versions/Variants/Releases

Version: An instance of a system which is functionally distinct in some way from other system instances. Version is the initial release or re-release of a configuration item associated with a complete compilation or recompilation of the item.

Variant: An instance of a system which is functionally identical but non-functionally distinct from other instances of a system.

Release: An instance of a system which is distributed to users outside of the development team.

1.3.7.2 Version Identification

Procedures for version identification should define an unambiguous way of identifying component versions.

There are following basic techniques for component identification

- Version numbering
- Attribute-based identification
- Change-oriented identification

Version Numbering

The actual derivation structure is a tree or a network rather than a sequence. Names are not meaningful. A hierarchical naming scheme is used .

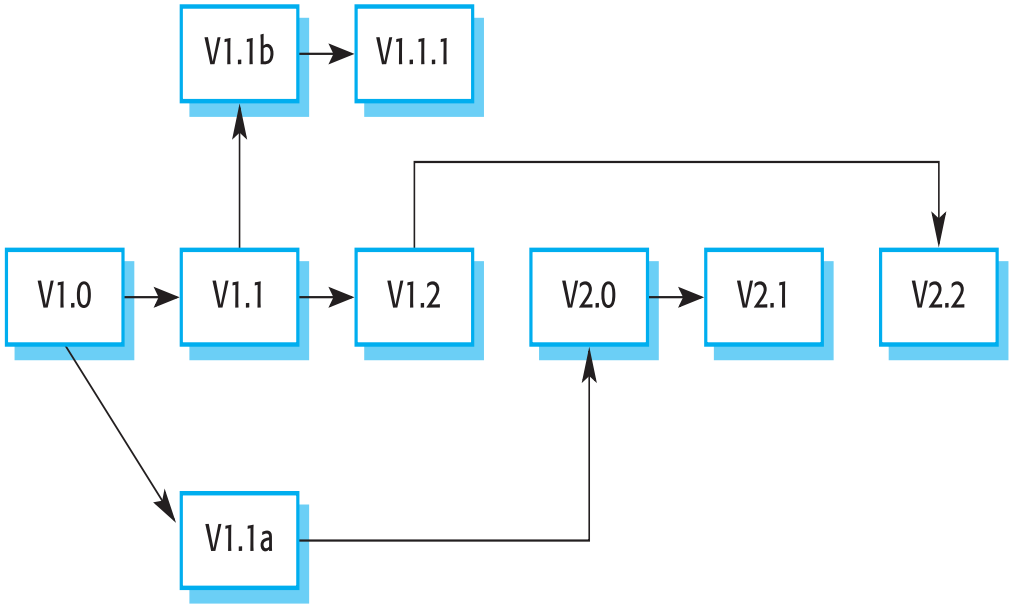


Figure 1.9: Version derivation structure

Attribute-based Identification

Attributes can be associated with a version with the combination of attributes identifying that version. Examples of attributes are Date, Creator, Programming Language, Customer, Status etc. This is more flexible than an explicit naming scheme for version retrieval; however, it can cause problems with uniqueness - the set of attributes have to be chosen so that all versions can be uniquely identified. In practice, a version also needs an associated name for easy reference. An important advantage of attribute-based identification is that it can support queries so that you can find 'the most recent version in Java' etc. The query selects a version depending on attribute values AC3D (language =Java, platform = XP, date = Jan 2003).

Change-Oriented Identification

- Integrates versions and the changes made to create these versions.
- Used for systems rather than components.
- Each proposed change has a change set that describes changes made to implement that change.

- Change sets are applied in sequence so that, in principle, a version of the system that incorporates an arbitrary set of changes may be created.

1.4 Motivation and Objective

A change is to be done in a software artifact due to errors occurring in the software or due to change in some software artifact. As a change is made in some artifact of the software, the change will induce errors in other software artifacts. Thus a change in an artifact induces side-effects in the software. Software developers have to put effort to find out the affected software artifacts due to a change in one artifact. Thus the development cost of software and time to develop software increases. This is the key problem of software industry. "*Software Artifacts based Change Controller*" solves this problem by organizing software artifacts in software repository. By using Software artifact based change controller the affected software artifacts are found out due to some change in some software artifact. Thus the software development effort and cost reduces

1.5 Organization of Thesis

This thesis report is divided into four chapters.

The introductory Chapter 1 defines the core motivation factor for the study of this peculiar approach, of software artifacts based change controller, as our research topic. In accordance to the requirements of our topic we provide the fundamental knowledge of concepts and terminologies involved in this process. It provides the introduction to the software artifacts, UML, software maintenance and change management.

Chapter 2 presents the literature review and problem statement. Major work carried out in this area has been discussed here. This chapter also focuses on research problem statement or the question related to the proposed methodology. This chapter draws a line of separation between our work and the existing work through gap analysis and provides a proper justification for it.

Chapter 3 explains the complete implementation details of the proposed system.

Chapter 4 discusses the results of the proposed system with the help of case study in order to verify the effort.

Finally, the Chapter 5 concludes the thesis work, includes summary of our contribution along with the scope for further research possibilities and its usage scenario. At last the work done in past by other intellectual researchers by means of references is given.

Chapter 2

Literature Review and Problem Statement

2.1 Mining Version Histories to Guide Software Changes

2.1.1 Overview of Approach

This approach [25] uses data mining to version histories in order to guide programmers along related changes. When a set of changes take place, it suggests and predicts likely further changes, shows up item coupling that is undetectable by program analysis, and prevent errors due to incomplete changes. This approach creates a ROSE prototype. Rose tool guides the programmer along related changes, with the following aims:

Suggest and Predict Likely Changes

Suppose a programmer just makes a change. What else the programmer has to change? Rose suggests considering further changes, as inferred from the version history.

Prevent Errors due to Incomplete Changes

If the programmer wants to commit the modifications without altering the suggested location, Rose would issue a warning, thus preventing errors due to incomplete changes.

Detect Coupling Undetectable by Program analysis

Since Rose operates uniquely on the version history, Rose is able to detect coupling between items that cannot be detected by program analysis including coupling between items that are not even programs.

This approach

- Uses data mining techniques to obtain association rules from version histories.
- Detects coupling between fine-grained program entities such as functions or variables thus increasing precision and integrating with program analysis.
- Thoroughly evaluates the ability to predict future or missing changes

Rose is not the first tool to leverage version histories. Before this tool, researchers have used history data to understand programs and their evolution [22], to detect evolutionary coupling between files [6] or classes [9], or to support navigation in the source code

[3]. To guide programmers, a number of tools have exploited textual similarity of log messages [1] or program code [4].

2.1.2 Processing Change Data

Figure 2.1 illustrates the basic data flow through Rose tool. The Rose server reads a version archive far left, groups the changes into transactions, mines the transactions for rules which describe implications between software entities: for example If fKeys[] is changed, then initDefaults() is typically changed, too. When the user changes some entity say, fKeys[] , the Rose client queries the rule set for applicable rules and makes appropriate suggestions for further changes, say, initDefaults().

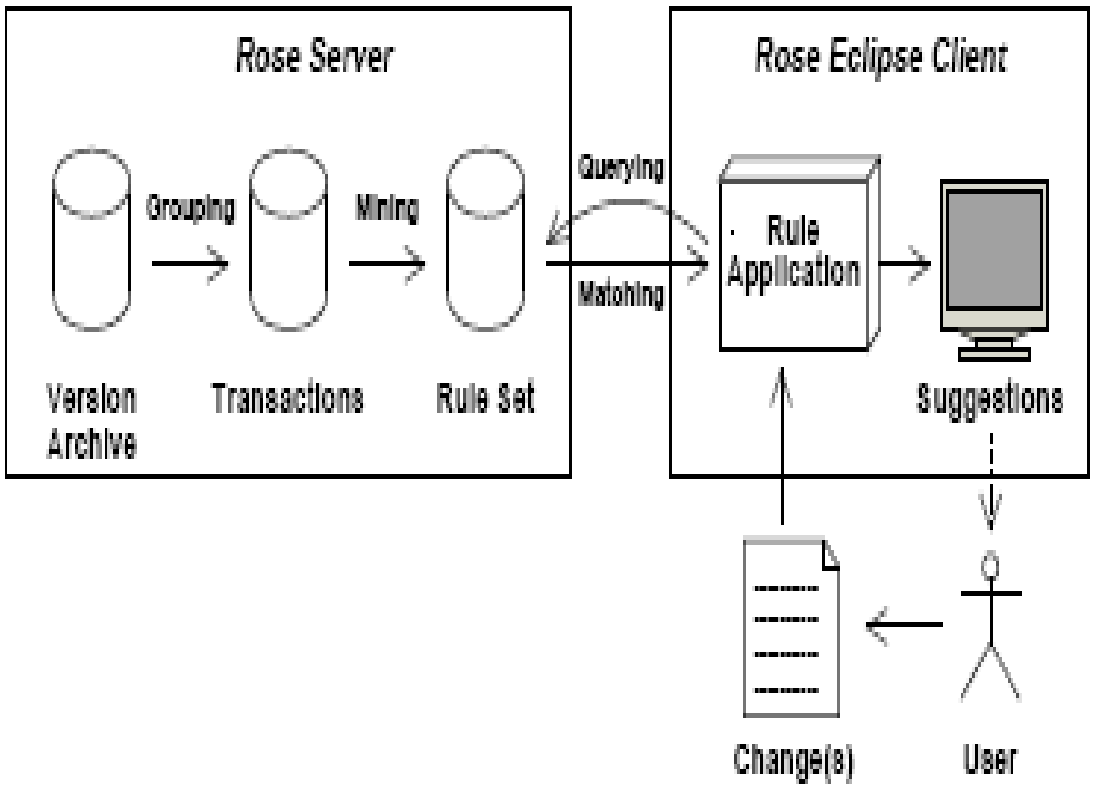


Figure 2.1: The data flow through Rose [25]

2.1.3 Grouping Changes to Transactions

Rose server retrieves changes and transactions from existing version archives typically from CVS archives, While CVS is popular, it has some weaknesses that require special

data cleaning [18].Rose follows the classical sliding window approach [12]: two subsequent changes δ_i and δ_{i+1} by the same author and with the same basis are part of one transaction if they are at most 200 seconds apart.

2.1.4 Conversion of Transactions into Rules

The aim of the Rose server is to mine rules from these transactions. An example of the rule is: {(Comp.java, field, fKeys [])}) ==> {(Comp.java, method, initDefaults ()), (plug.properties, file, plug.properties)}

This rule means that whenever the user changes the field fKeys [] in Comp.java, then user should also change the method initDefaults () and the file plug.properties. Rose uses the apriory algorithm [18] to compute association rules

2.2 How History Justifies System Architecture (or not)

This approach [26] uses revision history of a software system to convey important information about how and why the system evolved in time. The revision history also tells us which parts of the system are coupled by common changes. This evolutionary coupling can be compared with the coupling as imposed by the system architecture. Differences indicate anomalies which may be subject to restructuring. Rose prototype analyzes fine-grained coupling between software entities as indicated by common changes. It turns out that common changes are a good indicator for modularity, that evolutionary coupling should be determined between syntactical entities rather than files or module, and that common changes can indicate coupling between software entities and non-program artifacts that is unavailable to the analysis of a single version. The first work that leverages the product history to detect coupling within a system is the paper of Gall, Hajek, and Jazayeri [6]. They have used their CAESAR system to analyze the coupling within a large telecommunication switching system, and found that the history of 20 releases can indeed show up coupling within a system. A similar work was conducted by Bieman, Andrews and Yangon classes [9], using 39 releases of a commercial object oriented system

In a software product, two entities are said to be coupled if a change in an entity A implies a change in another entity B i.e. B depends on A. good software design attempts to minimize and encapsulate dependencies such that future changes induce as few further changes as possible. The basic idea is that common changes of entities indicate evolutionary dependencies: whenever the database schema was changed, the sqlquery() method was altered, too. The more frequently entities have been changed together, the stronger they are coupled.

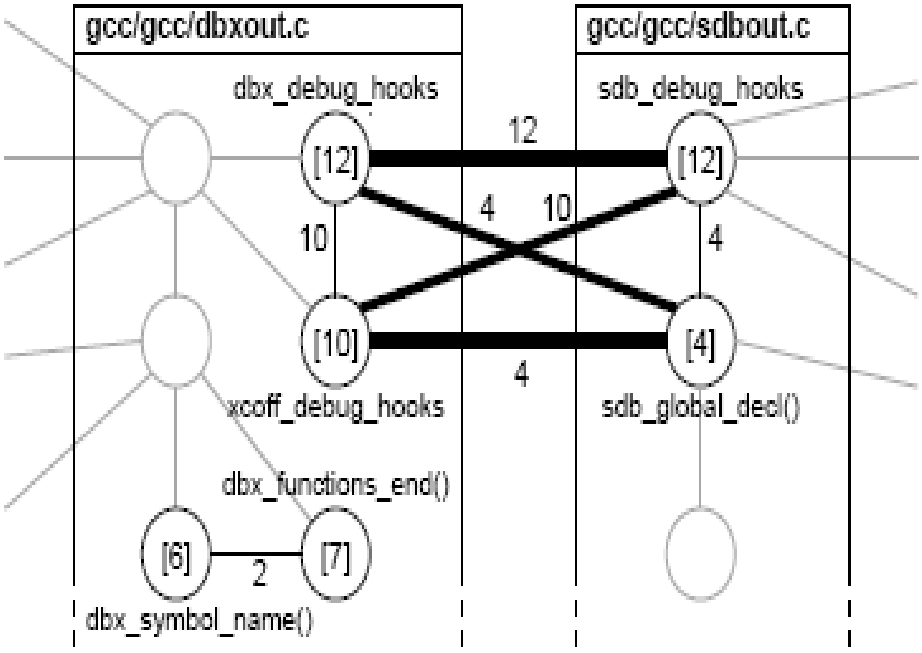


Figure 2.2: Evolutionary coupling in GCC [26]

As a simple example, consider Figure 2.2, visualizing the evolutionary coupling of some program entities in the GNU Compiler Collection (GCC) as obtained from the GCC CVS history using Rose prototype. Two files dbxout.c and sdbout.c issue debugging symbols in DBX and SDB format, respectively. Both files contain some program entities, depicted as vertices—variables such as dbx_debug _hooks in dbxout.c and methods such as sdb _global_decl (). The numbers in brackets show how frequently the entity has been changed over the revision history of GCC—xcoff_debug_hooks, for instance, has been changed ten times. Two entities are related by edges if they ever have been changed at the same time i.e. they are coupled by a common change in the GCC CVS archive. The

number associated with each edge indicates how often the related entities have been changed together. So it can be observed that

• In all 12 cases where dbx_debug _hooks was changed, so was sdb_debug_ hooks, and vice versa.

• In all 4 cases where sdb_global_decl() was changed, so were the other debug hooks variables—in both files.

• dbx _functions_end() and dbx _symbol_name() have been changed together, but never with an entity in sdbout.c.

In general, using the revision history as additional knowledge source allows for an improved assessment of software architectures:

Detecting Coupling between Non-Program Entities

Think of a simple coupling between a database schema and an SQL query method. Whenever the former is changed, the later must be adapted, too. Detecting such a coupling from the software product's contents would require a very specific analysis that knows about syntax and semantics of the database as well as the query method. However, this coupling can easily be established from revision history.

Detecting Coupling without Program Analysis

Revision histories are available for almost every non-trivial software project. To establish coupling between stored artifacts typically files, it is not required to analyze the artifact contents. A light weight analysis can easily associate textual changes to syntactic entities, such that coupling can be established on a finer-grained basis say, coupling between functions stored in a file.

Comparing Evolutionary and Specified Coupling

Evolutionary coupling from revision history and analytical coupling from program analysis can be determined independent of each other, and thus compared with each other. In the ideal case, every evolutionary coupling should also be an analytical coupling, thus justifying the system architecture. Mismatches indicate possible targets for restructuring. This approach relates changes to individual program entities like functions, methods, and attributes. It thus detects fine-grained coupling between these entities, allowing for a much better understanding of commonalities and anomalies

2.3 When Do Changes Induce Fixes?

As a software system evolves, programmers make changes that sometimes cause problems. This approach [11] analyzes CVS archives for fix-inducing changes. Fix inducing changes are the changes that lead to problems, indicated by fixes. This approach automatically locates fix-inducing changes by linking a version archive such as CVS to a bug database such as BUGZILLA. In a first investigation of the MOZILLA and ECLIPSE history, it turns out that fix-inducing changes show distinct patterns with respect to their size and the day of week they were applied

Software histories are mined frequently in order to detect patterns that help us understanding the current state of the system. Unfortunately, not all changes in the past have been beneficial. Any bug database will show a significant fraction of problems that are reported some time after some change has been made. This approach is an attempt to identify those changes that caused problems. The basic idea is as follows:

- Start with a bug report in the bug database, indicating a fixed problem.
- Extract the associated change from the version archive, thus finding out the location of the fix.
- Determine the earlier change at this location that was applied before the bug was reported.

This earlier change is the one that caused the later fix. Such a change is known as fix-inducing.

2.3.1 How to Extract Data from Version and Bug Archives

For analysis, all changes and all fixes of a project are needed. This data can be obtained from version archives like CVS and bug tracking systems like bugzilla. A CVS archive contains information about changes: Who changed what, when, why, and how? A change δ transforms a revision r_1 to a revision r_2 by inserting, deleting, or changing lines. Several changes $\delta_1, \ldots, \delta_n$ form a transaction t if they were submitted to CVS by the same developer, at the same time, and with the same log message, i.e., they have been made with the same intention, e.g. to fix a bug or to introduce a new feature. As CVS records only individual changes to files, these changes are grouped to transactions with a sliding time window approach

A CVS archive also lacks information about the purpose of a change: did it introduce a new feature or did it fix a bug? Although it is possible to identify such reasons solely with log messages [2], both CVS and bugzilla for this step because this increases the precision of our approach.

A bugzilla database collects bug reports that are submitted by a reporter with a short description and a summary. After a bug has been submitted, it is discussed by developers and users who provide additional comments and may create attachments. After the bug has been confirmed, it is assigned to a developer who is responsible to fix the bug and finally commits her changes to the version control archive. Bugzilla also captures the status of a bug, e.g., UNCONFIRMED, NEW, ASSIGNED, RESOLVED, or CLOSED and the resolution, for ex. FIXED, DUPLICATE, or INVALID. Details on the lifecycle of a bug can be found in the bugzilla documentation [24].

2.3.2 Identifying Fixes

In order to locate fix-inducing changes, this is necessary whether a change is a fix. A common practice among developers is to include a bug report number in the comment whenever they fix a defect associated with it Murphy [3] as well as Fischer, Pinzger, and Gall [15, 14] exploited this practice to link changes with bugs.

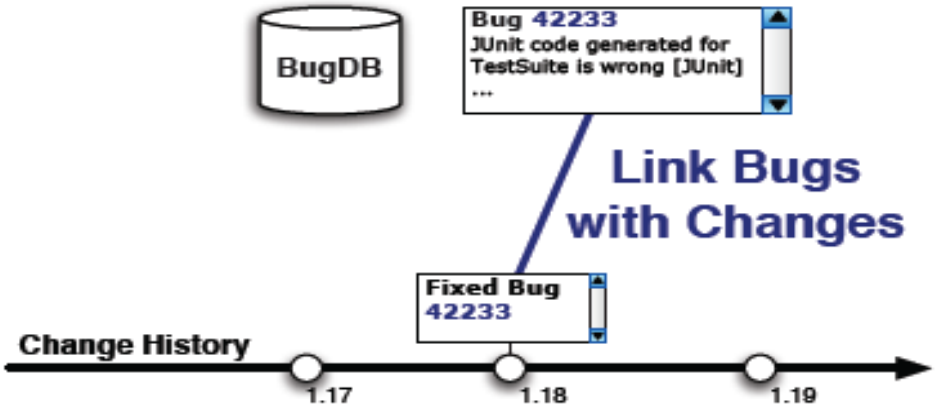


Figure 2.3: Link transactions to bug reports [11]

Figure 2.3 sketches the basic idea of this approach. In this work, these techniques are refined by assigning every link (t, b) between a transaction t and a bug b two independent levels of confidence: a syntactic level, inferring links from a CVS log to a bug report, and

a semantic level, validating a link via the bug report data. These levels are used to decide which links shall be taken into account

2.3.3 Locating Fix Inducing Changes

A fix-inducing change is a change that later gets undone by a fix. Suppose that a change δ ε t, which is known to be a fix for bug b (thus a link (t, b) must exist), transforms the revision r_1 = 1.17 of Foo.java into r_2 = 1.18 as shown in figure 2.4 i.e., δ introduces new lines to r_2 or changes and removes lines of r_1.First lines L that have been touched by δ in r_1 are detected. These are the locations of the fix. To locate them, the CVS diff command is used. In example, we assume that line 20 and 40 have been changed and line 60has been deleted, thus the fix locations in r_1 are L = {20; 40; 60}.

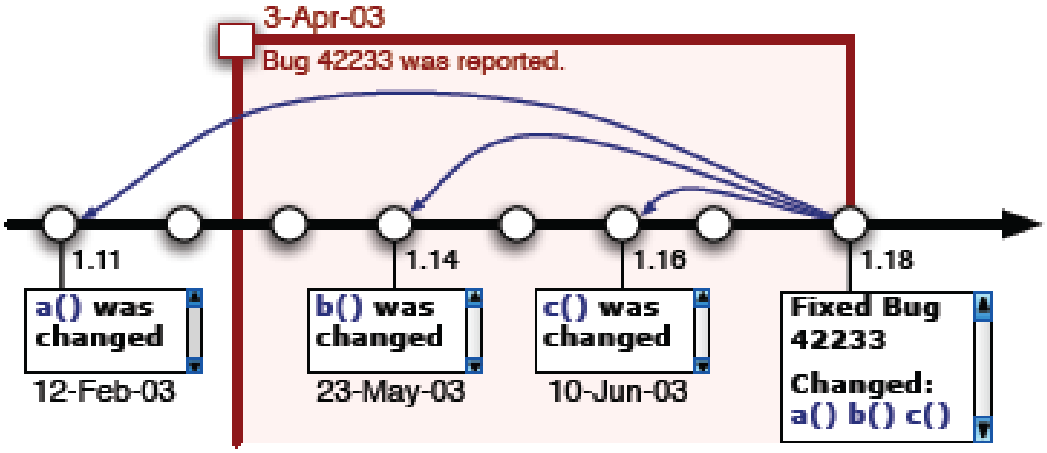


Figure 2.4: Locate fix-inducing changes for bug [11]

Next, the CVS annotate command for revision r_1 = 1.17 is used because this was the last revision without the fix; in contrast, revision r_2 = 1.18 already contains the applied fix. The annotations prepend each line with the most recent revision that touched this line. Additionally, CVS includes the developer and the date in the output. The CVS annotate command is only reliable for text files, thus all files that are marked as binary in the repository. The output is scanned and for each line l ε L the revision r_0 that annotates line l is taken. These revisions are candidates for fix-inducing changes. (r_0, r_2) is added to the candidate set S, which is S = {(1.11, 1.18); (1.14, 1.18); (1.16, 1.18)}.From this set, it is made of removal of pairs (r_a, r_b) for which it is not possible that r_a induced the fix r_b—for instance, because r_a was committed to CVS after the bug fixed by r_b has been reported. In

particular, such a pair (r_a, r_b) is a suspect if r_a was committed after the latest reported bug linked with the revision r_b. Suspect changes could not contribute to the failure observed in the bug report. In Figure 2.4 the pairs (1.14, 1.18) and (1.16, 1.18) are examples of suspects.

2.4 Mining a Change-Based Software Repository

Although state-of-the-art software repositories based on versioning system information are useful to assess the evolution of a software system, the information they contain is limited in several ways. Versioning systems such as CVS or Subversion store only snapshots of text files, leading to a loss of information: The exact sequence of changes between two versions is hard to recover. This approach [20] represents an alternative information repository which stores incremental changes to the system under study, retrieved from the IDE used to build the software. Then this change based model of system evolution is used to assess classical versioning system repositories

The nature of information found in software repositories determines what we can infer from it. Conversely, information missing from a software repository hampers the quality of the research we perform: What is stored in a repository is of prime importance. However, another characteristic limits the choice of source code repositories: the number of available case studies [21].Even if Estublier et al. write in [10], that one of the next steps for versioning systems to break the assumption of language independence, this is not yet the case in practice. These assumptions are too weak for researchers to perform precise research on source code evolution.

This approach creates a software repository designed to store a maximal amount of information about an evolving piece of software. In particular, this approach does not use a versioning system, but built from the ground up a change-based software repository which fetches domain-specific information from an Integrated Development Environment (IDE). Being change-based means that the evolution of a software system is not modelled as a sequence of versions anymore: Using an IDE allows us to store, as first-class citizens, the actual changes which were performed on the system to obtain its latest version. This model better matches the actual evolution of a system since how developers actually change the system can be reproduced.

2.4.1 Limitations of Versioning Systems

Versioning systems have two shortcomings which limit the amount of information we can recover from them: They are file-based and snapshot-based [21] [19].A file-based versioning system versions files and lines. Text files are a poor medium to perform precise source code analysis. A heavy pre-processing must be performed to raise the abstraction level beyond file and lines. Snapshot-based versioning systems store updates in their repositories as deltas between two snapshots of the system. Versioning systems only rely on notifications from the developer, they cannot guarantee small deltas. An exception is the approach presented by Schneider et al. in [13], in which changes are automatically committed to a secondary repository. However the interval at which this is done is not specified and the source code might not be in a compliable state. The size of the deltas between the retained version grows much larger. Sampling is a common enough practice that it was included as a requirement of Kenyon [8], a framework for evolution analysis.

2.4.2 Change-based Software Repositories

In a change-based repository the history of a software system is not viewed as a sequence of versions, but rather as the sum of change operations which were necessary to bring the system to its actual state. These operations can not be deduced with enough precision by differencing two arbitrary program versions [16]. Instead they are recovered by monitoring the IDE usage of programmers while they are building the software. Building a change-based repository involves the following steps:

Program Representation

How a software system in a change-based repository is represented to match the problem domain as closely as possible.

Change Operations

What constitutes a change operation on a software system, and how to abstract from the lower-level details when they are not needed.

Data Retrieval

How to retrieve change operations from an evolving system, using an IDE. An approach can be adopted such by using an IDE, which can be open to external contributors by means of a plug-in mechanism [3] [27].

2.5 Gaps in Existing Approaches

The brief comparison of the proposed methodology with the existing work is given in form of following table 2.1. The comparative study below helps to find out certain shortcomings in the existing methodologies with comparison of the proposed work.

Existing Approaches	Comparison Results
Mining Version Histories to Guide Software Changes	This approach detects coupling between fine-grained program entities such as functions or variables. This approach suggests the impact of changes in an entity of one module to another entity in the same module The proposed approach is not only limited to the impact of changes in an entity of one module to another entity in the same module but the proposed approach finds other affected software artifacts due to change in an artifact.
How History Justifies System Architecture or not	This approach uses revision history of a software system to convey important information about how and why the system evolved in time. The revision history tells which parts of the system are coupled by common changes. A file-based versioning system versions files and lines. Text files are a poor medium to perform precise source code analysis. A heavy pre-processing must be performed to raise the abstraction level beyond file and lines: parsing

	every version of the system to build a model of it, and then linking these models together to obtain the overall history of the system in terms of versions. The proposed approach use association between the software artifacts. So heavy preprocessing is not required as in file versioning systems.
When Do Changes Induce Fixes?	This approach analyzes CVS archives for fix-inducing changes. Fix inducing changes are the changes that lead to problems, indicated by fixes. This approach automatically locates fix-inducing changes by linking a version archive such as CVS to a bug database. The proposed approach is an attempt to find the affected software artifacts due to change in any software artifact.
Mining a Change-Based Software Repository	This approach creates a change-based software repository which fetches domain-specific information from an IDE. Using an IDE allows to store the actual changes which were performed on the system to obtain its latest version. This approach finds the impact of change in one unit to other unit. But the proposed approach not only finds the affected code due to change in one software artifact but also other software artifacts.

Table 2.1: Gap Analysis between the proposed approach and existing approaches

The loopholes between the proposed approach and existing approaches are the motivation factors of proposed work.

2.6 Concise Problem Statement

The proposed approach finds the affected software artifacts due to change in any software artifact. The artifacts are stored in the software repository .The artifacts are then associated with each other. A GUI tool has been developed for storage of software artifacts and association between software artifacts. If a user wants to find the affected software artifacts due to change in any software artifact, the tool allows the user to select a particular software artifact and results the affected software artifacts due to change in selected software artifact.

2.7 Justification

The proposed approach not only finds the affected unit due to some change made in one unit but also finds the affected requirement and design. The proposed approach does not use to store version history. So heavy pre-processing is not required .The gap analysis in the previous section shows that the loop holes in the existing work are removed by this approach. The proposed method allows the user to find the affected software artifacts due to change in any software artifact. To best of our knowledge there is no such approach which directly gives the affected software artifacts due to change in any software artifact based on VB.Net platform and SQL server. The tool developed, is efficient and saves effort of the user to find the affected software artifacts due to any change of any type in any software artifacts.

Chapter 3

Proposed Methodology

Change management is the necessity of the software. Software need to be changed due to changes in software artifacts. Artifacts may be software requirements, software design and software code. Whenever an artifact changes, it will affect other software artifacts. This approach finds which artifact will be changed due to change in an artifact.

3.1 System Overview

The system developed is called as "Software Artifacts based Change Controller". Two main works performed by the system are following:

- Software Repository population
- Retrieval of affected software artifacts due to change in an artifact

First software repository is populated by inputting software artifacts such as software requirement, software design, code and test cases. After that if a software artifact is changed, the system will retrieve the affected software artifacts due to changes in one software artifact.

3.2 Experimental Setup

Software artifacts storage module is developed in VB.Net and SQL Server 2005. The repository can be populated with any other standard databases available like MS Access or Oracle. Software artifacts associator module is developed in vb.net and SQL Server 2005.The input is software artifact that is either software requirement or software design or software code stored in software repository. Software design is represented by UML class diagram. UML Class diagram can be developed with the help of Rational Rose software tool. To model UML diagrams, some software for modeling is required. In this work the software used is the Rational Rose, which is a product of IBM Corporation. In Rational Rose UML class diagram can be modeled and stored in a file of MDL file format.

3.3 Working of the System and Implementation

The structure of the system is given in figure 3.1.The key functionality of the system lies in the Associator.

Associator performs following two functions:

- Storage of software artifacts in the software repository
- Retrieval of the affected software artifacts due to change in a software artifact.

Associator has two main modules to perform the above functions:

- Software Artifact Locator
- Software Artifact Retrieval module

3.3.1 Software Artifact Locator

Software artifact locator is the module of the system used to populate the software artifacts repository. SAL (Software artifact locator) stores the software artifacts such as project, software requirements, software design, code and test in the software repository. SAL also links the software artifacts for ex. SAL links the software requirements with software design, software design with code and code with test. This linkage is necessary so that Artifact Linker will reflect the changed software artifacts due to change in some software artifact

SAL is further divided into two parts:

- Artifact Populator
- Artifact Linker

3.3.1.1 Artifact Populator

Artifact populator is the module used to store the software artifacts in the software repository. Artifact populator populates the software requirements, software design, code and test in the software repository. Artifact populator is further subdivided into following sub parts to perform the above functions

- Project Storage Module
- Requirement Storage Module
- Design Storage Module
- Test Storage Module

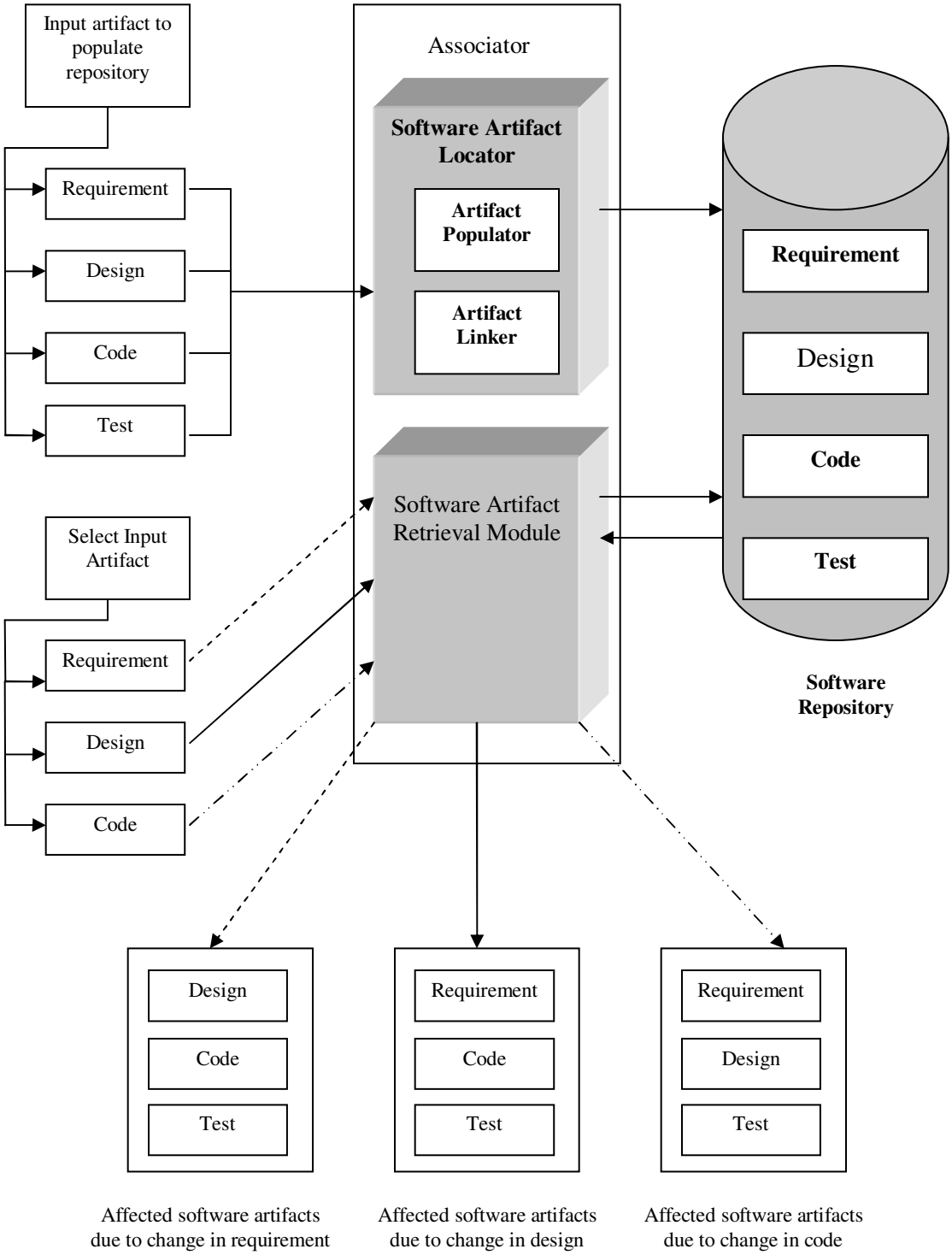


Figure 3.1: Software Artifacts based Change Controller

- Code Storage Module

The various sub modules of artifact populator is shown in the figure 3.2.

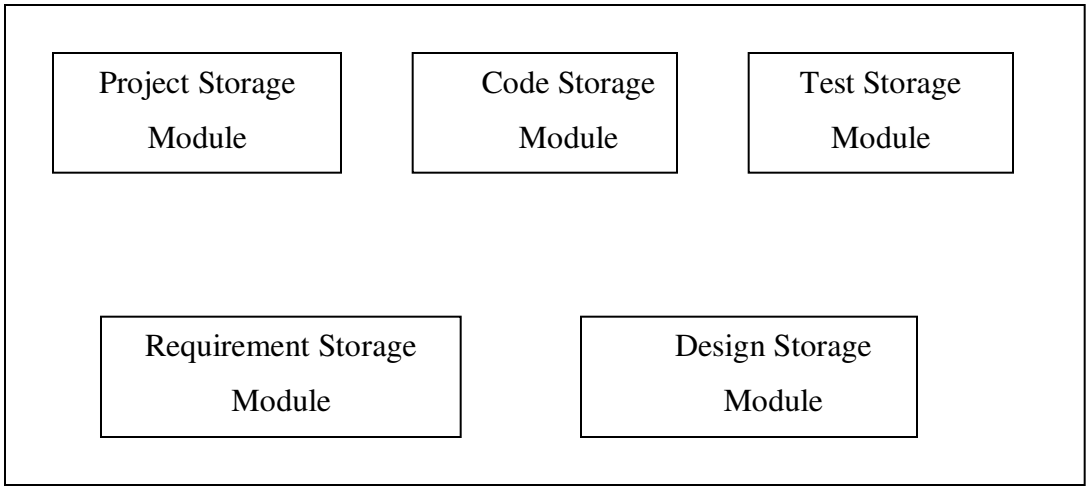


Figure 3.2: Artifact Populator

Project Storage Module

PSM (Project storage module) stores information about the project in the software Artifacts repository. Whenever a project is to be stored in the Software repository, a corresponding Project id (pid) is generated in the project table. The schema diagram of project table is shown in the following diagram.

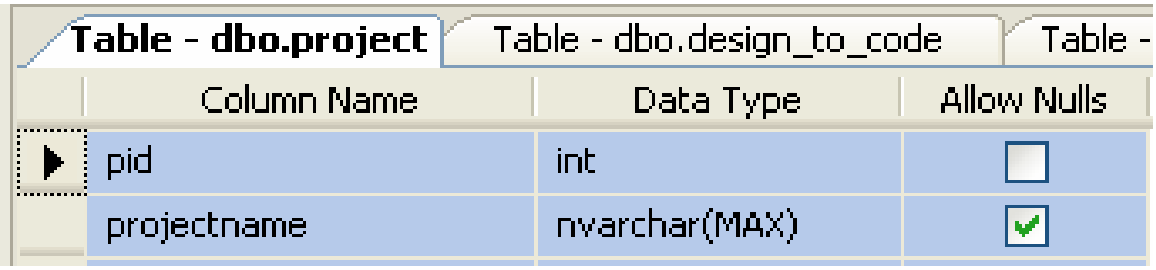


Figure 3.3: Project schema

Requirement Storage Module

RSM (Requirement storage module) stores the software requirements in the software repository. The requirement table consists three fields: pid, requirement id, requirement

requirement id: A unique identifier to identify the requirement

requirement: name of the requirement

pid: id of the project corresponding to the requirement

The requirement schema diagram is shown in figure 3.4

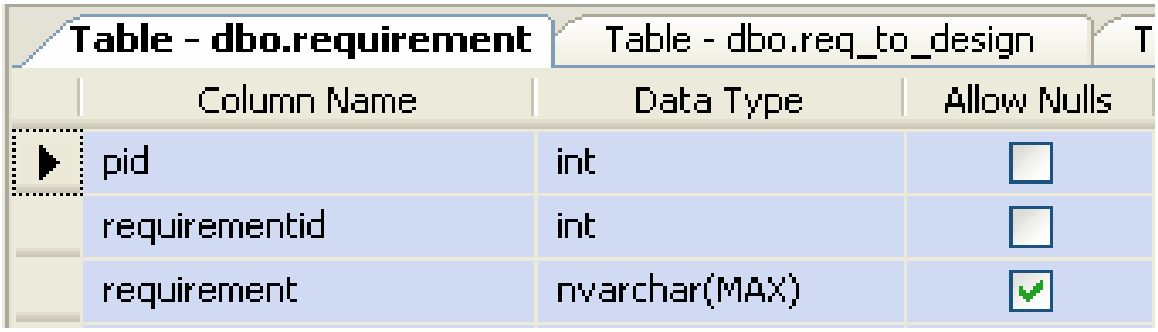


Figure 3.4: Requirement schema

Whenever a requirement is to be stored in software repository, a requirement id is generated automatically. Requirement storage module stores requirement, requirement id, and pid(project id) corresponding to the project of that requirement in the requirement table.

Design Storage Module

DSM (Design Storage Module) stores the software design in the Software Artifacts Repository. Software design is generally in the Mdl format. Software design is constructed in Rational Rose, a product of IBM Corporation. Design schema is shown in figure 3.5

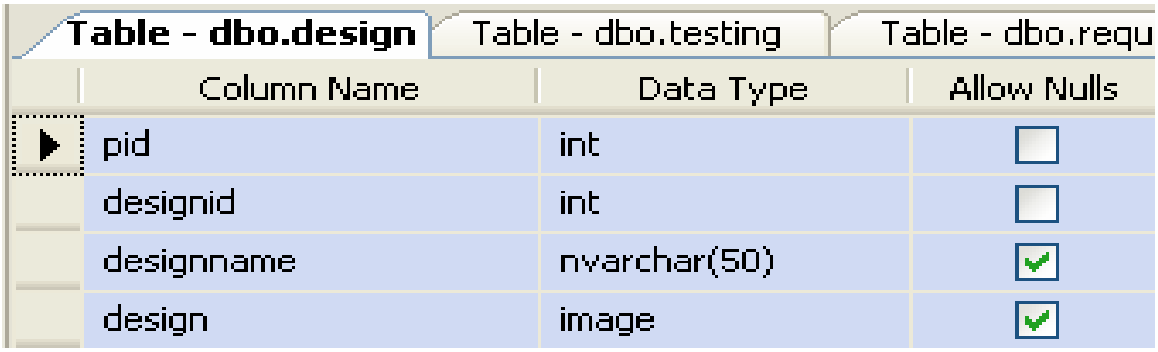


Figure 3.5: Design schema

The design schema consists four fields: pid, design id, design name and design.

pid: the project id corresponding to the design to be stored

designid: a unique identifier to identify the design

design name: name of the design

design: a design file corresponding to the design

Whenever a design is to be stored in software repository, a design id is generated automatically. Design storage module stores design name, design, design id and project id corresponding to the project of that design in the design table.

Code Storage Module

CSM (Code storage module) stores the software code in the software repository. Coding schema is shown in figure 3.6.

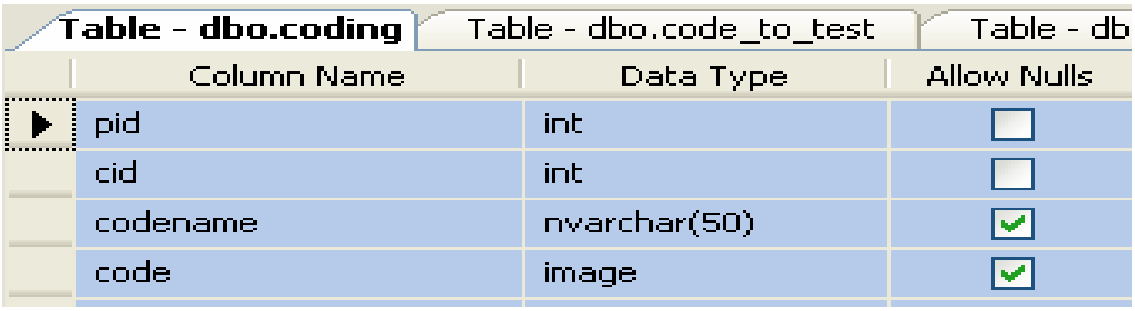


Figure 3.6: Coding schema

The coding schema consists four fields: pid, cid, code name and code.

pid: the project id corresponding to the code to be stored

cid: a unique identifier to identify the code.

codename: name of the code-class

code: a code-class file corresponding to the code.

Whenever a code-class is to be stored in software repository, a code id (cid) is generated automatically. Code storage module stores code, code name, cid (code id) and pid (project id) corresponding to the project of that code-class in coding table.

Test Storage Module

TSM (Test storage module) stores the software test in the software repository. Testing schema is shown in figure 3.7. The testing schema consists four fields: pid, tid, test name and test file.

pid: the project id corresponding to the test to be stored

tid: a unique identifier to identify the test.

test name: name of the test

test file: a test-file corresponding to the test.

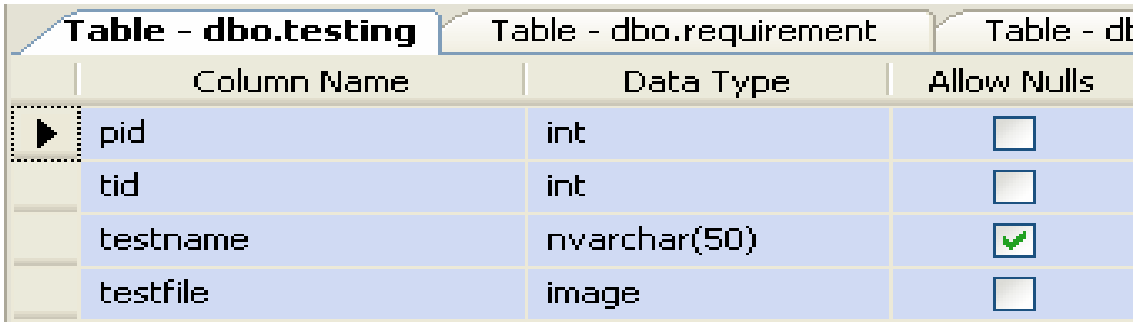


Figure 3.7: Testing schema

Whenever a testing file is to be stored in software repository, a test id (tid) is generated automatically. Test storage module stores testing file, test name, tid (test id) and pid(project id)corresponding to the project of that testing file in testing table.

3.3.1.2 Artifact Linker

Artifact Linker is the subpart of software artifact locator. The Artifact linker is shown in figure 3.8.

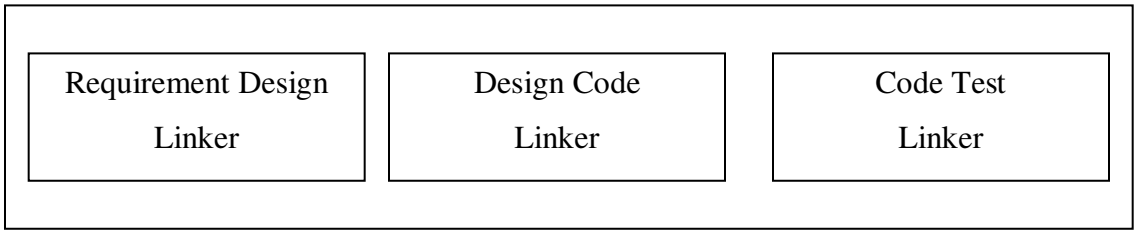


Figure3.8: Artifact Linker

Artifact linker links various software artifacts. Artifact linker links the following:

- Software requirement to software design
- Software design to software code
- Software code to software test

The above functions are performed by the sub-modules of Artifact linker. The various sub modules of artifact Linker are following:

- Requirement Design Linker
- Design Code Linker
- Code Test Linker

Requirement Design Linker

One requirement may be associated with one or more design. In the same way one or more design may be associated with one requirement. The above illustration is supported by the figure3.9:

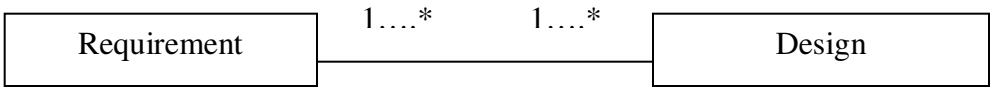


Figure 3.9: Requirement Design Association

Requirement design linker links the various requirements with their corresponding design. Req_to_design is a table which stores requirement id and design id. Req_to_design schema is shown in figure3.10.

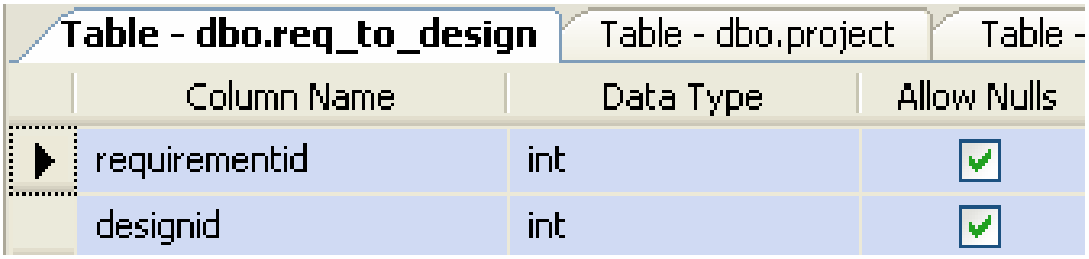


Figure 3.10: Req _to_design schema

Whenever a requirement is to be linked with the design, Requirement design linker stores requirement id of the requirement and the design id of the corresponding design in the req_to_design table.

Design Code Linker

One or more code is linked to a design. This illustration is supported by the figure 3.11:

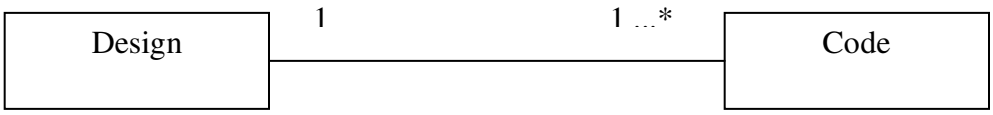


Figure 3.11: Design Code Association

Design code linker links various code with their corresponding design.

Design_to_code is a table which consists two fields: design id and cid. Figure 3.12 shows design _to_code schema.

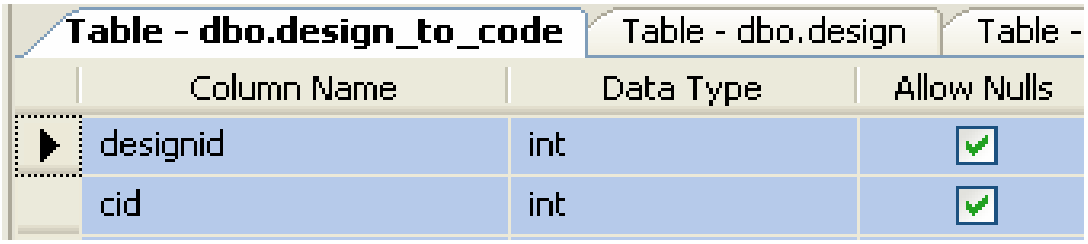


Figure 3.12: Design _to_code schema

Whenever a design is to be linked with the corresponding code, design code linker stores design id of the design and cid(code id) of the corresponding code classes in the design_to_code table.

Code Test Linker

One code may be associated with one or more test cases. In the same way one or more test may be associated with one code. The above illustration is supported by figure 3.13:

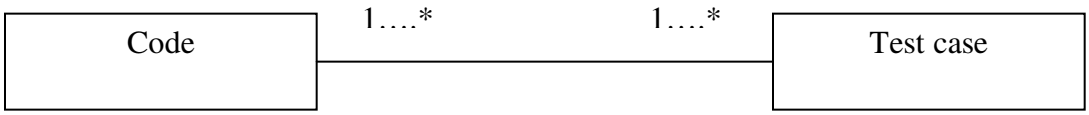


Figure 3.13: Code Test Association

Code test linker links various codes with their corresponding test files.

Code_to_test is a table which consists two fields: cid (code identity)and tid(test id). Figure 3.14 shows code_to_test schema.

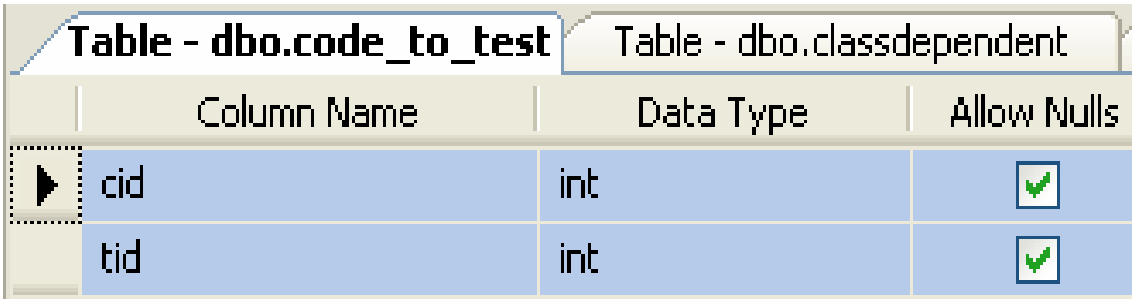


Figure 3.14: Code_to_test schema

\Whenever a code is to be linked with the test case, code test linker stores cid of the code and tid(test id) of the corresponding in the code_to_test table.

3.3.2 Software Artifact Retrieval Module

Software artifact retrieval module retrieves the affected software artifacts due to change in a software artifact. Input to this module is a software artifact. The output of this module is the affected software artifacts due to change in a software artifact. Whenever a change is made in requirement, it will affect design, code and test cases. Whenever a change is made in design, it will affect the requirement, code and test cases. Whenever a change is made in code, it will affect requirement, design and test cases. A class may be inherited from other class also. So whenever a change is made in a code- class, it will affect other code-classes also. Software artifact retrieval module consists three modules:

- Software Requirement impact module
- Software Design impact module
- Software code impact module

The various sub-modules of software artifact retrieval module are shown in figure 3.15.

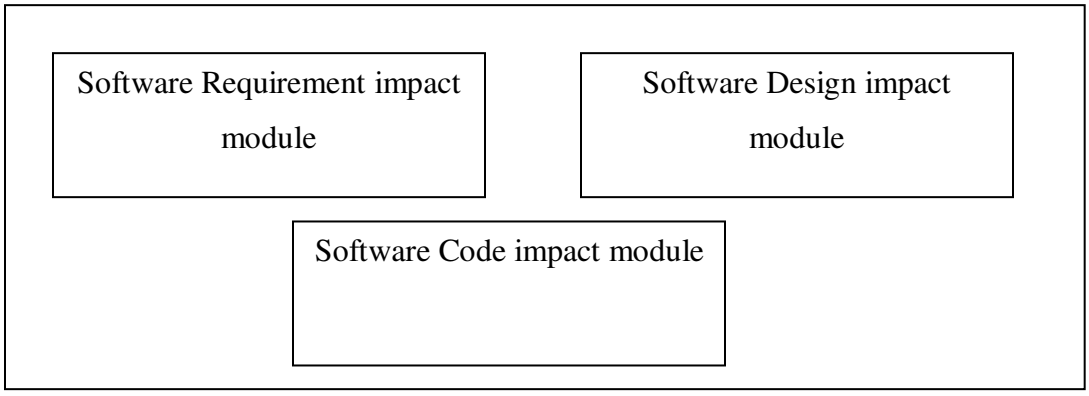


Figure 3.15: Software Artifact Retrieval module

Software Requirement Impact Module

When a change is made in software requirement, software requirement impact module associates following:

- the requirement id of the requirement with corresponding design id,
- design id with cid(code id)
- cid with tid

Thus software requirement impact module is able to find design ids of the affected designs, cids of the affected codes and tids of the affected test files.

After that software requirement impact module retrieves the following artifacts:

- designs related to design ids
- codes related to cids
- tests related to tids

Software Design Impact Module

When a change is made in software design, software design impact module links following:

- design id of the design with corresponding requirement id
- design id of the design with corresponding cid(code id)
- cid with tid.

Thus software design impact module is able to find requirement ids of the affected requirements, cids of the affected codes and tids of the affected test files.

After that software design impact module retrieves the following artifacts:

- Requirements related to requirement ids
- codes related to cids
- tests related to tids.

Software Code Impact Module

When a change is made in software code, software code impact module links following:

- cid(code id)of the code with corresponding design id
- design id of the design with corresponding requirement id
- cid with tid.
- cid of the dependent code.

Thus software code impact module is able to find design id of the affected design, requirement ids of the affected requirement, cids of dependent code-classes and tids of the affected test file.

After that software code impact module retrieves the following artifacts

- Requirements related to requirement ids

- Designs related to design ids.
- Dependent Codes related to cids
- tests related to tid.

Chapter 4

Experimental Results

As change management is the need of software development. Most of the approaches used till now find the effects of changes of software code to other unit. There has not been any effort of finding the effects of changes in one software artifact such as design to other software artifacts affected such as requirement, coding and testing. This approach finds the artifacts to be changed when change takes place in any software artifact. To support our work we present a case study.

Case Study

Consider a hotel management system. The requirements, corresponding design files, code files and test files are as follows:

Requirements	Design file(.mdl)	Code file(.java)	Test file(.doc)
1.Customer makes payment either by cheque or cash.	Order_payment	Customer_order Order Payment Treatment Credit Cash cheque	Order Order Money Treatment Money Money money
2.Customer requires medical facility in hotel.	Order_payment	Customer_order Order Payment Treatment Credit Cash cheque	Order Order Payment Treatment Credit Cash cheque
3.There should be attendance system	Attendance	Person_at Attendance	Attendance Attendance

of customer and employee.		Morning_att Evening_att	Attendance Attendance
4. Customer wants facility for booking room.	Accomdation	Customer_room room	Room Room
5.Customer enquires by phone or personally.	Enquiry	Customer_enq Enquiry Party enquiry Facility enquiry Accomdationenquiry Payment enquiry	Enquiry Enquiry Enquiry Enquiry Enquiry Enquiry
6.Accountant tracks the record of daily income and expenditure.	Income expenditure Item Accommodation Attendance Order_payment	Account Item Customer_room Room Person_at Attendance Morning_att Evening_att Customer_order Order Payment Treatment Credit Cash cheque	Account Item Room Room Attendance Attendance Attendance Attendance Order Order Payment Treatment Credit Cash cheque
7.An item not available in the hotel is purchased by the management	Item	Item	Item

8.Inventory adds the item depending upon manufacturing_date, expiry_date.	Inventory	Inventory Manufacturing_date Expiry_date	Inventory Date Date
9. Inventory deletes the item depending upon manufacturing_date, expiry_date.	Inventory	Inventory Manufacturing_date Expiry_date	Inventory Date Date
10. Manager recruits and terminates the employee and employee information is updated.	Employee-Management Employee_info	Employee_recruit Manager Employee_info	Employee Manager Employee
11. Hotel provides party facility.	Enquiry	Customer_enq Enquiry Party enquiry Facility enquiry Accomdationenquiry Payment enquiry	Enquiry Enquiry Enquiry Enquiry Enquiry Enquiry

Table4.1: Requirements, Design files, coding files and test files of Hotel Management System

This approach has two options:

1. New Project: This option stores project, requirements, design files coding files, test files inn the database. Association between requirements & design, design & code, code & test are also stored by this option.

2. Change Project: This option allows the user to select a software artifact such as requirement, design, coding, testing This option retrieves the artifacts to be affected if a change takes place in some software artifact.

4.1 New Project

This option stores project, requirements, design files coding files, test files in the database. Association between requirements & design, design & code, code & test are also stored by this option. This approach uses 2 figures a & b corresponding to one software artifact. Figure a shows the procedure of storing the artifact or association in the software repository. Figure b shows Table after inserting the artifact or association.

4.1.1 Storing the Project

Figure 4.1a: Adding project hotel management system in repository

Figure 4.1a shows that when a user clicks New Project in File menu, a form is opened where the name of project is entered Proceed next option allows the user to update the project table entries as shown in figure 4.1b.As Proceed Next button is clicked, a pid of project is generated.

Figure 4.1b: Updated table project in repository

4.1.2 Storing the Requirement

Requirements can be stored in the software repository after storing project in software repository. Add Requirement button allows to store the requirements in the software repository as shown in figure 4.2 a.

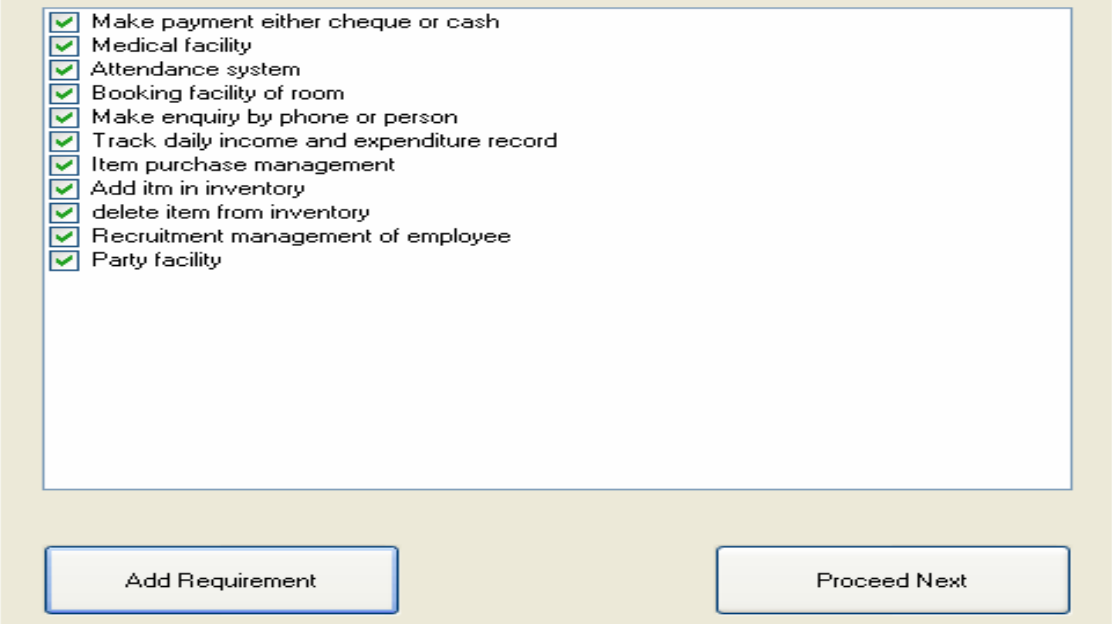


Figure 4.2a: Adding the requirements in the repository

As the requirement is added in the software repository, a requirement id is generated and the fields of table requirement are updated with requirement name, requirement id and corresponding project id.

Table - dbo.requirement | Table - dbo.project | Table - dbo.req_to_design

	pid	requirementid	requirement
▶	20	47	Make payment either cheque or cash
	20	48	Medical facility
	20	49	Attendance system
	20	50	Booking facility of room
	20	51	Make enquiry by phone or person
	20	52	Track daily income and expenditure record
	20	53	Item purchase management
	20	54	Add itm in inventory
	20	55	delete item from inventory
	20	56	Recruitment management of employee
	20	57	Party facility
*	NULL	NULL	NULL

Figure 4.2b: Updated table requirement in repository

4.1.3 Storing the Design

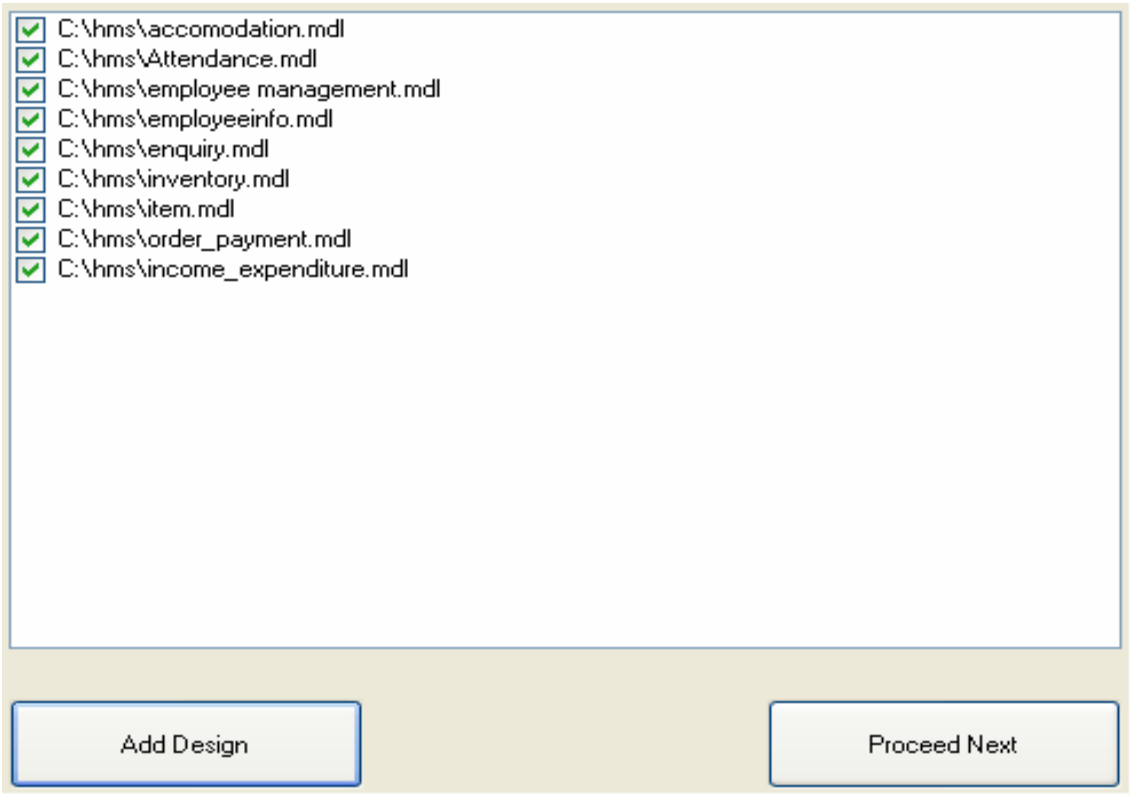


Figure 4.3a: Adding the design files in the repository

Design in mdl can be stored in the software repository after storing requirement in software repository. Add Design button allows to store the design in the software repository as shown in figure 4.3 a. As the design file is added in the software repository, a design id is generated and the fields of table design are updated with design name, design id and corresponding project id as shown in figure 4.3b.

Table - dbo.req_to_design | Table - dbo.coding | **Table - dbo.design**

	pid	designid	designname	design
▶	20	41	accomodation.mdl	<Binary data>
	20	42	Attendance.mdl	<Binary data>
	20	43	employee management.mdl	<Binary data>
	20	44	employeeinfo.mdl	<Binary data>
	20	45	enquiry.mdl	<Binary data>
	20	46	inventory.mdl	<Binary data>
	20	47	item.mdl	<Binary data>
	20	48	order_payment.mdl	<Binary data>
	20	49	income_expenditure.mdl	<Binary data>
*	NULL	NULL	NULL	NULL

Figure 4.3b: Updated table design in the repository

4.1.4 Storing the Code

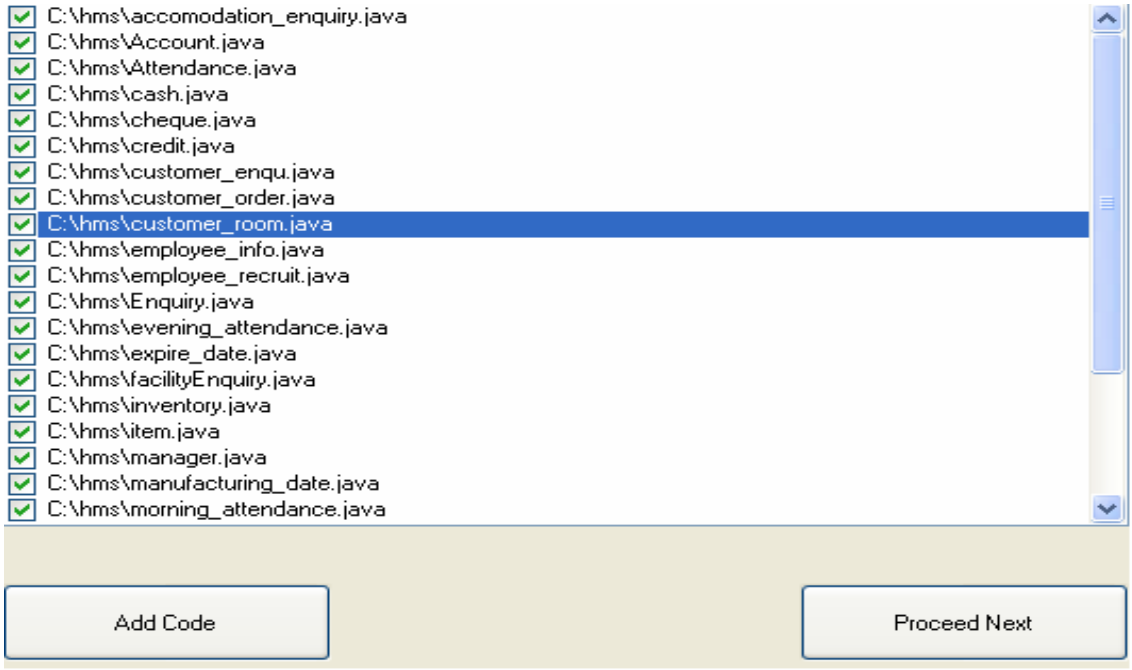


Figure 4.4a: Adding the code files in the repository

Table - dbo.req_to_design | **Table - dbo.coding** | Table - dbo.design

	pid	cid	codename	code
▶	20	32	accomodation_enquiry.java	<Binary data>
	20	33	Account.java	<Binary data>
	20	34	Attendance.java	<Binary data>
	20	35	cash.java	<Binary data>
	20	36	cheque.java	<Binary data>
	20	37	credit.java	<Binary data>
	20	38	customer_enqu.java	<Binary data>
	20	39	customer_order.java	<Binary data>
	20	40	customer_room.java	<Binary data>
	20	41	employee_info.java	<Binary data>
	20	42	employee_recruit.java	<Binary data>
	20	43	Enquiry.java	<Binary data>
	20	44	evening_attendance.java	<Binary data>
	20	45	expire_date.java	<Binary data>
	20	46	facilityEnquiry.java	<Binary data>
	20	47	inventory.java	<Binary data>
	20	48	item.java	<Binary data>
	20	49	manager.java	<Binary data>
	20	50	manufacturing_date.java	<Binary data>
	20	51	morning_attendance.java	<Binary data>
	20	52	Order.java	<Binary data>

Figure 4.4b: Updated table coding in repository

Code in java can be stored in the software repository after storing design in software repository. Add code button allows to store the code files in the software repository as shown in figure 4.4 a. As the code file is added in the software repository, a cid is generated and the fields of table code are updated with code name, cid and corresponding project id. The updated table coding is shown in figure 4.4b after adding the code in repository.

4.1.5 Storing the Test Files

Test files can be stored in the software repository after storing coding in software repository. Add test button allows storing the test files in the software repository as shown in figure 4.5a. As the test file is added in the software repository, a tid is generated and the fields of table testing are updated with test name, tid and corresponding project

id. The updated table testing is shown in figure 4.5b after adding the test files in repository.

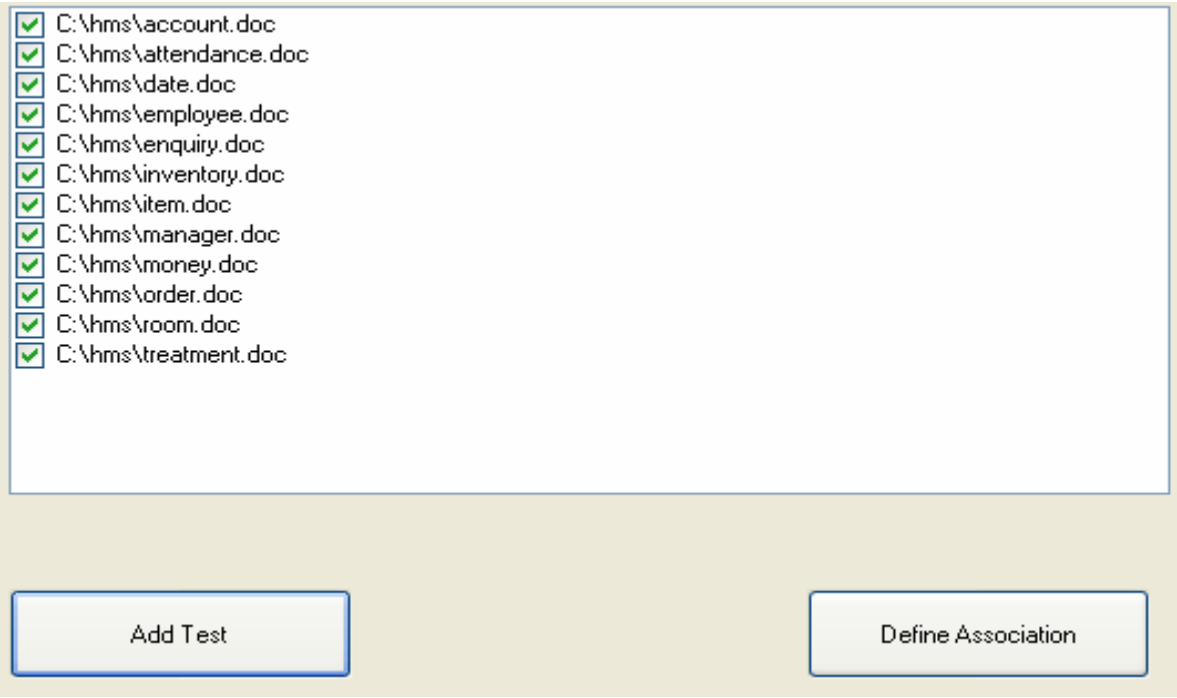


Figure 4.5a: Adding the test files in repository

Table - dbo.testing | Table - dbo.requirement | Table - dbo.project

	pid	tid	testname	testfile
▶	20	22	account.doc	<Binary data>
	20	23	attendance.doc	<Binary data>
	20	24	date.doc	<Binary data>
	20	25	employee.doc	<Binary data>
	20	26	enquiry.doc	<Binary data>
	20	27	inventory.doc	<Binary data>
	20	28	item.doc	<Binary data>
	20	29	manager.doc	<Binary data>
	20	30	money.doc	<Binary data>
	20	31	order.doc	<Binary data>
	20	32	room.doc	<Binary data>
	20	33	treatment.doc	<Binary data>
*	NULL	NULL	NULL	NULL

Figure 4.5b: Updated table testing in repository

4.1.6 Storing Association between Requirement and Design

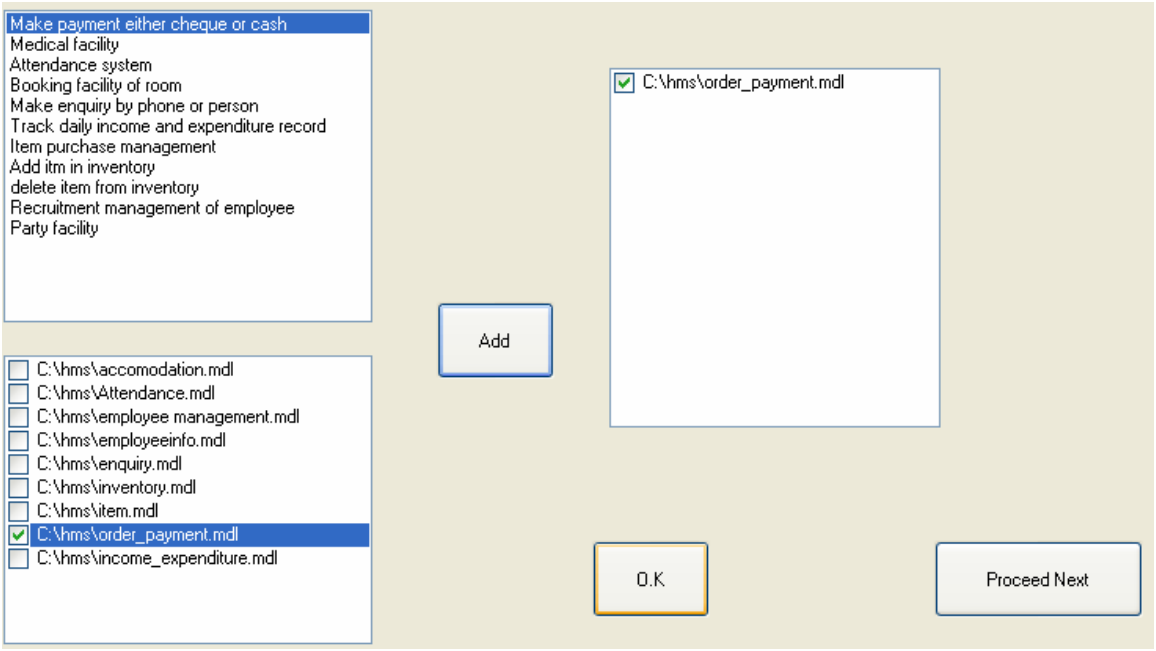


Figure 4.6a: Associating the requirement to design in repository

Table - dbo.req_to_design | Table - dbo.

	requirementid	designid
▶	47	48
	48	48
	49	42
	50	41
	51	45
	52	41
	52	42
	52	47
	52	48
	52	49
	53	47
	54	46
	55	46
	56	43
	56	44
	57	45
*	NULL	NULL

Figure 4.6b: Updated table requirement to design

A requirement may be associated to one or more design files. A user can select one requirement and the corresponding designs as shown in figure 4.6a and associate the requirement to corresponding design files. After associating all the requirements with their corresponding design files, user presses Proceed Next button of figure 4.6a.As Proceed Next button is pressed, requirement to design table is filled with requirement id of requirement and the design id of corresponding design as shown in figure 4.6b.

4.1.7 Storing Association between Design and Code

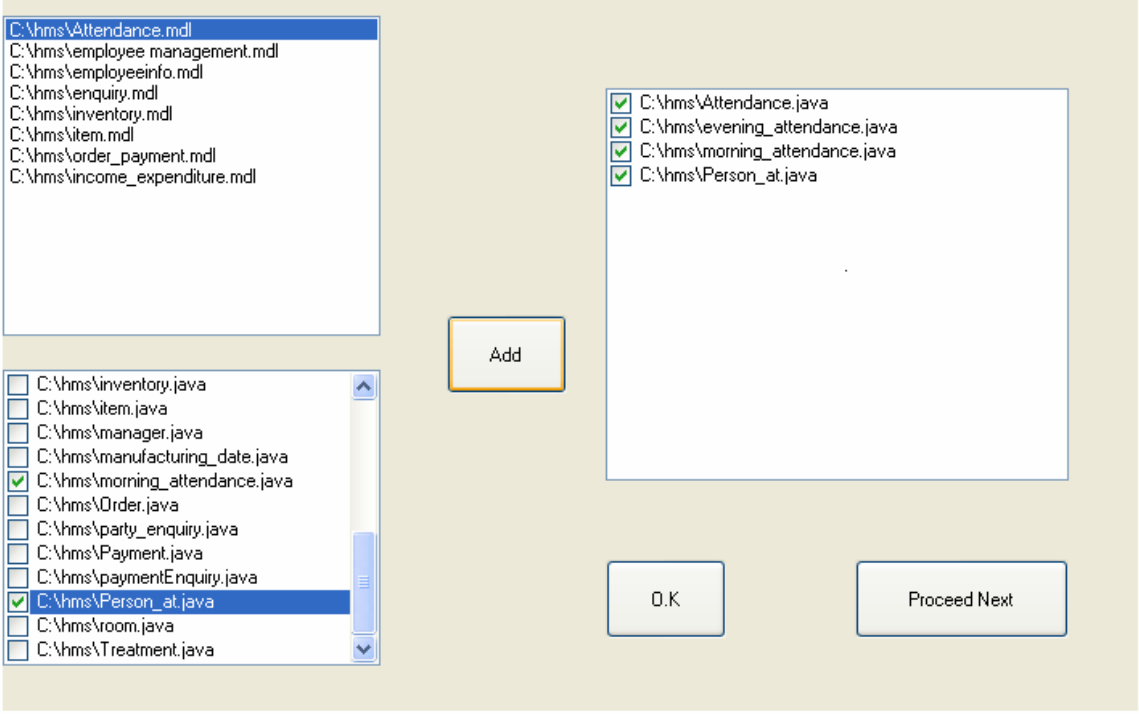


Figure 4.7a: Associating design to code

A design may be associated with one or more code files. A user can select one design file and the corresponding code files as shown in figure 4.7a and associate the design to corresponding code files. After associating all the designs with their corresponding code files, user presses Proceed Next button of figure 4.7a.As Proceed Next button is pressed, design to code table is filled with design id of design and the codeid of corresponding code as shown in figure 4.7b.

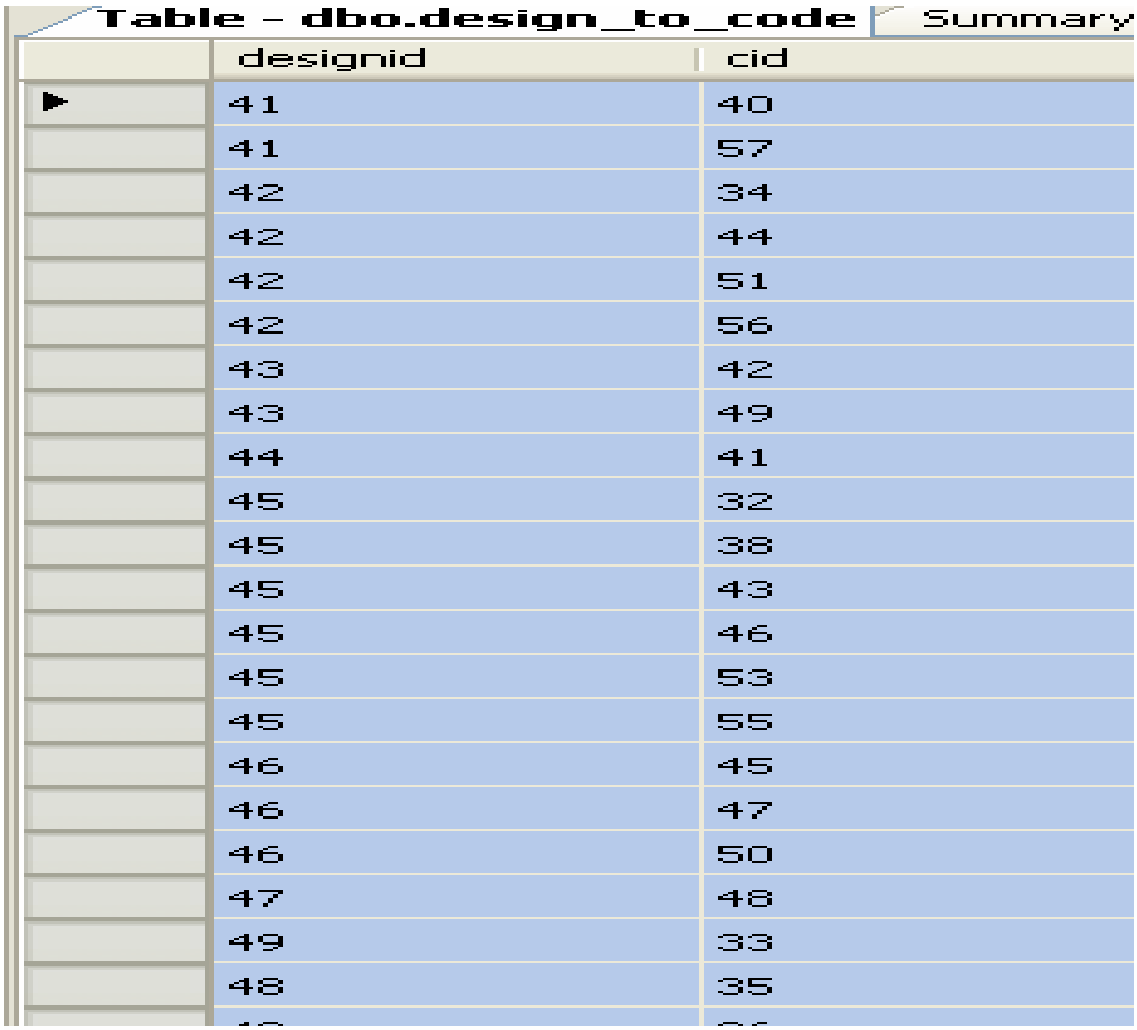


Table - dbo.design_to_code | Summary

designid	cid
41	40
41	57
42	34
42	44
42	51
42	56
43	42
43	49
44	41
45	32
45	38
45	43
45	46
45	53
45	55
46	45
46	47
46	50
47	48
49	33
48	35

Figure 4.7b: Updated table design to code

4.1.8 Storing Association between Code and Test

A code file may be associated with one or more test files. A user can select one code file and the corresponding test files as shown in figure 4.8a and associate the code files to their corresponding test files. After associating all the code files with their corresponding test files, user presses Finish button of figure 4.8a.As Finish is pressed, code to test table is filled with code id of code and the test id of corresponding test as shown in figure 4.8b.

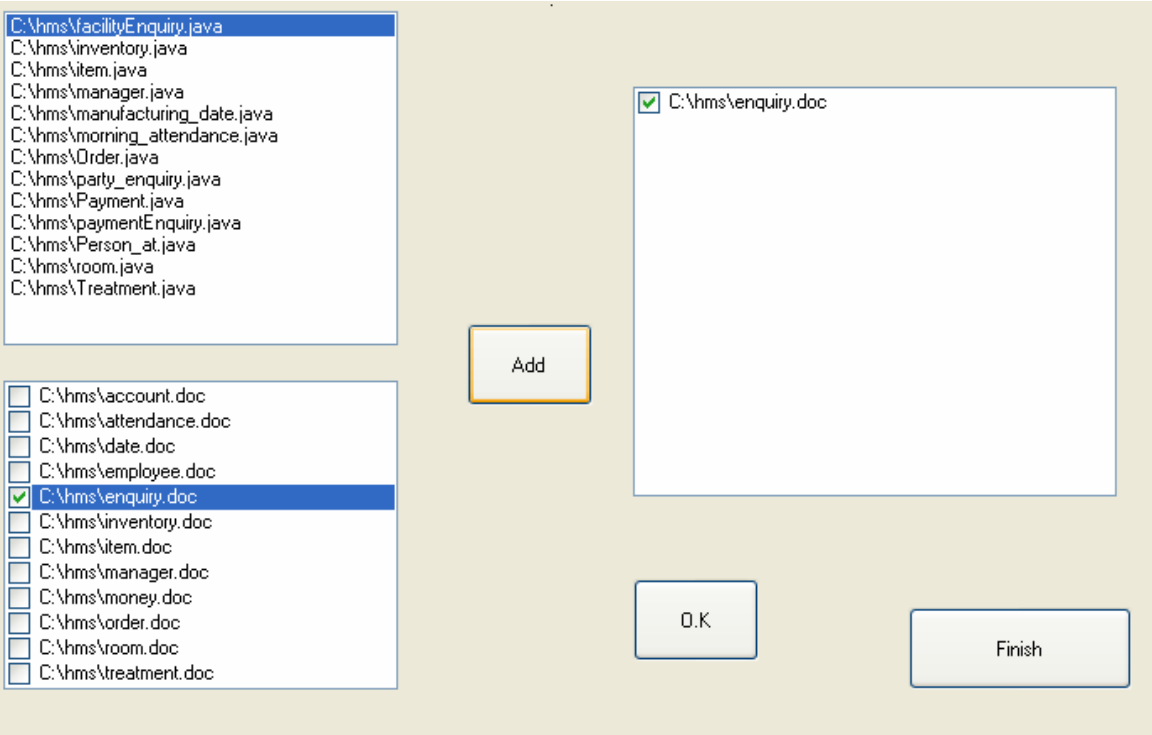


Figure 4.8a Associating code to test

Table - dbo.code_to_test | Table - dbo.c

cid	tid
32	26
33	22
34	23
35	30
36	30
37	30
38	26
39	31
40	32
41	25
42	25
43	26
44	23
45	24
46	26
47	27
48	28
49	29
50	24
51	23
52	31
53	26

Figure 4.8b: Updated table code to test

4.2 Change Project

Change Project option of file menu allows the user to select an artifact and finds out the artifacts to be affected if some change takes place in the selected artifact. The selected artifacts by user may be requirement, design and code.

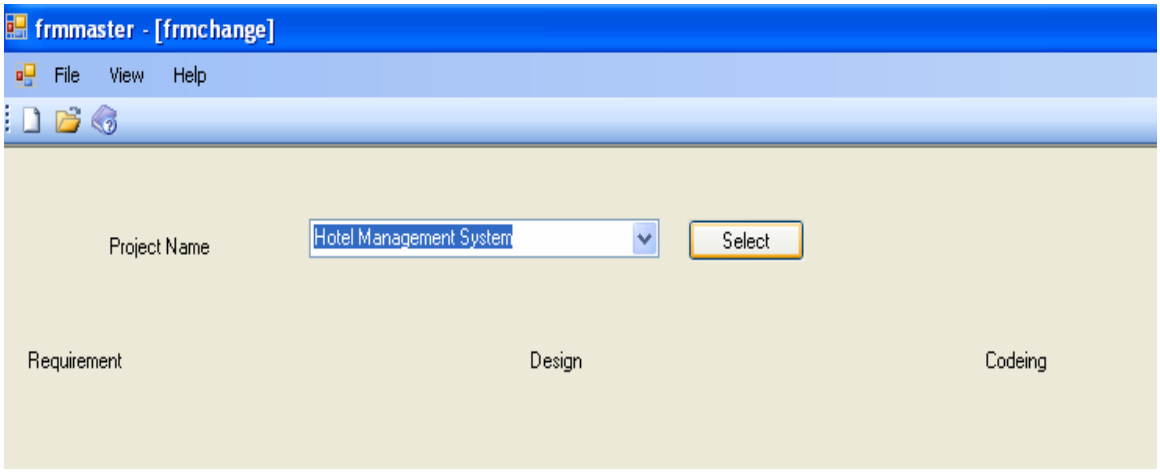


Figure 4.9: Selecting the project

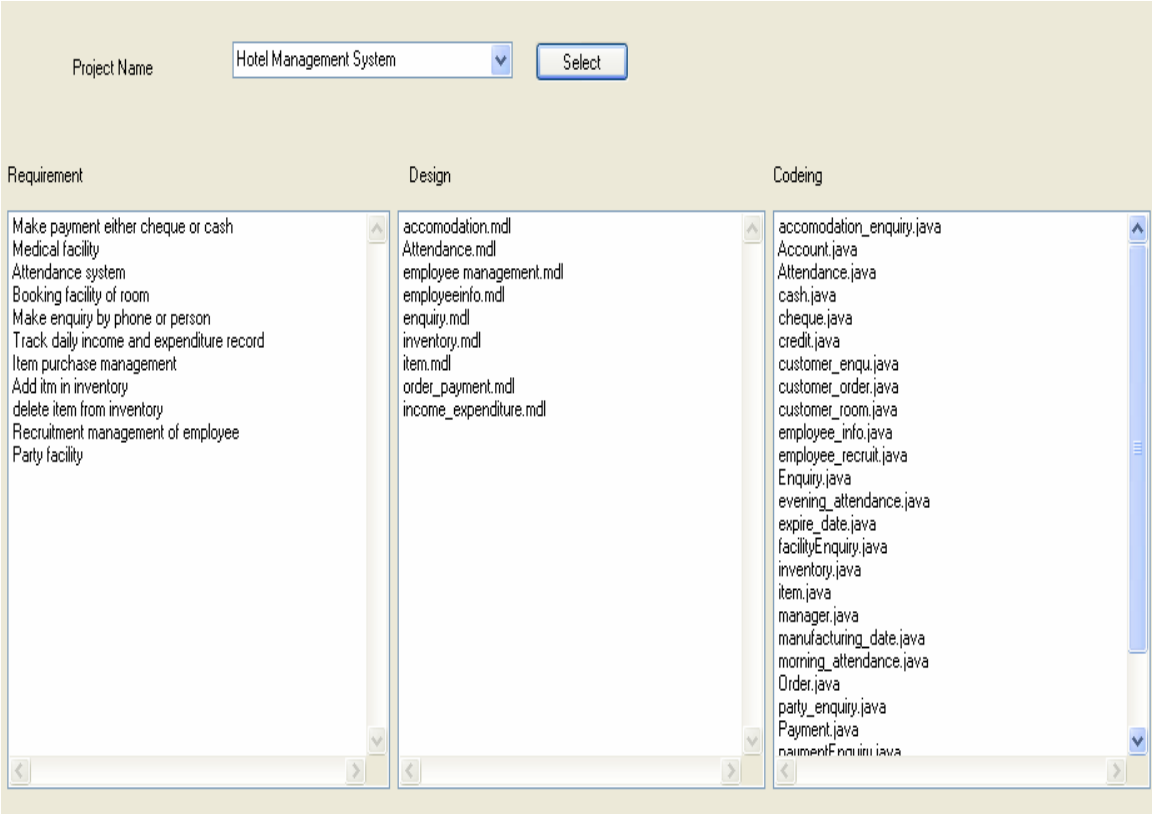


Figure 4.10: Interface for selecting Software Artifacts

4.2.1 Affected Artifacts due to Requirement Change

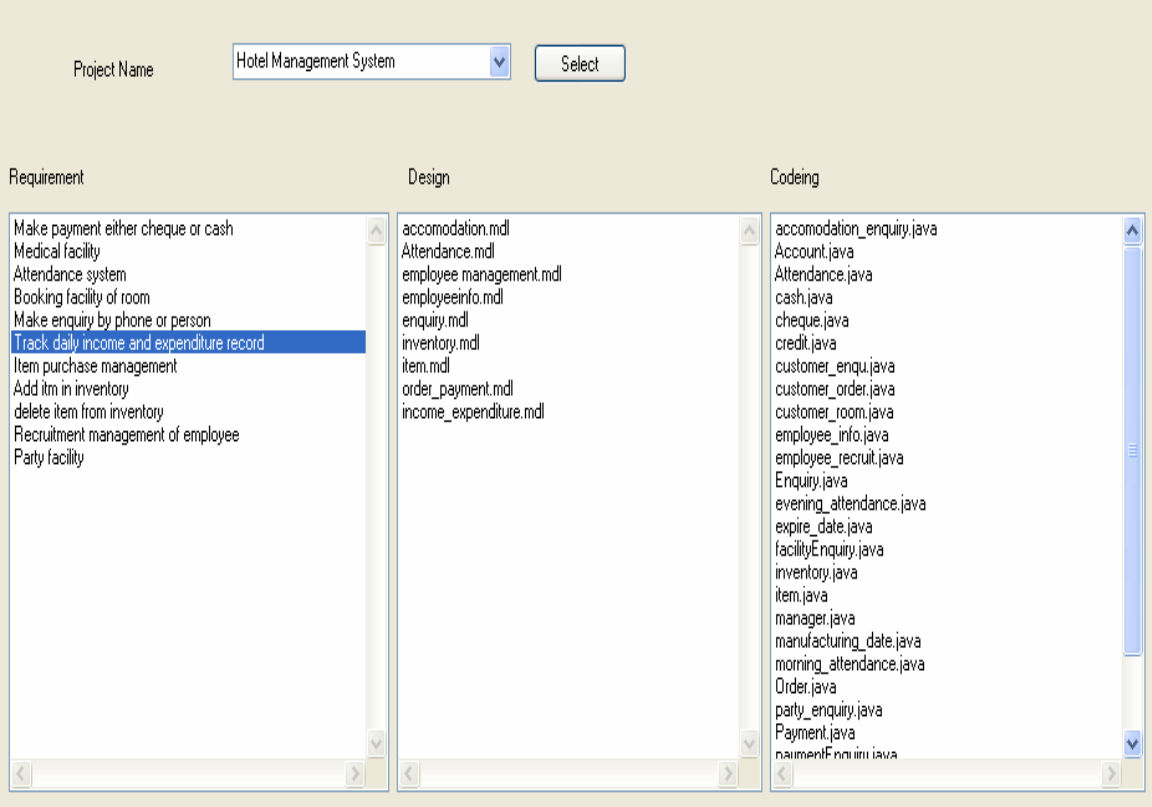


Figure 4.11a: Selected requirement

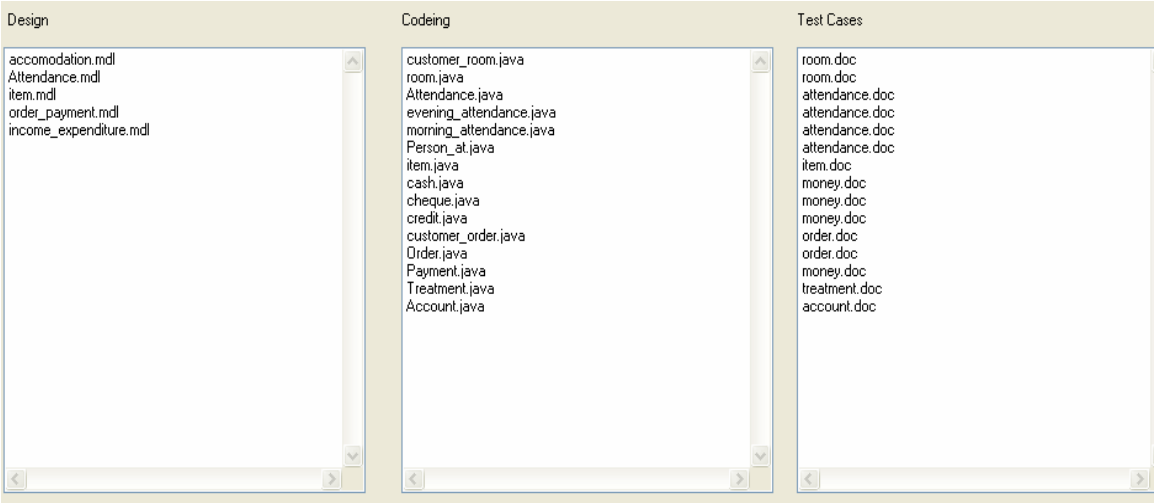
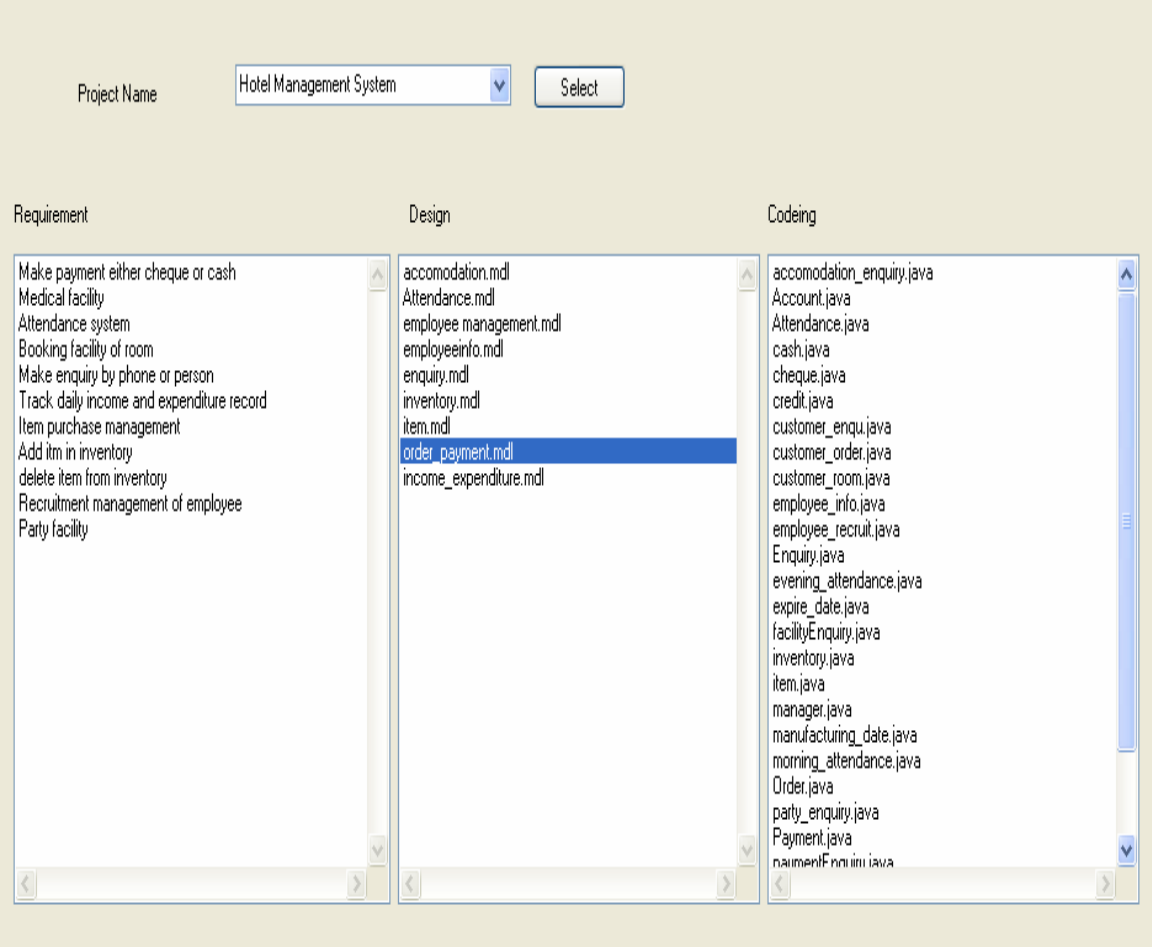


Figure 4.11b: Design, coding and test files affected due to change

The user can select any requirement whose affect to be found, from requirement window as shown in figure 4.11a. As user clicks the requirement, the corresponding design, code and test files affected are displayed as shown in figure 4.11b.

4.2.2 Affected Artifacts due to Design Change



4.12a: Selected design file

The user can select any design file, whose affect to be found, from design window as shown in figure 4.12a.As user clicks the selected design, the corresponding requirement, code and test files affected are displayed as shown in figure 4.12b.

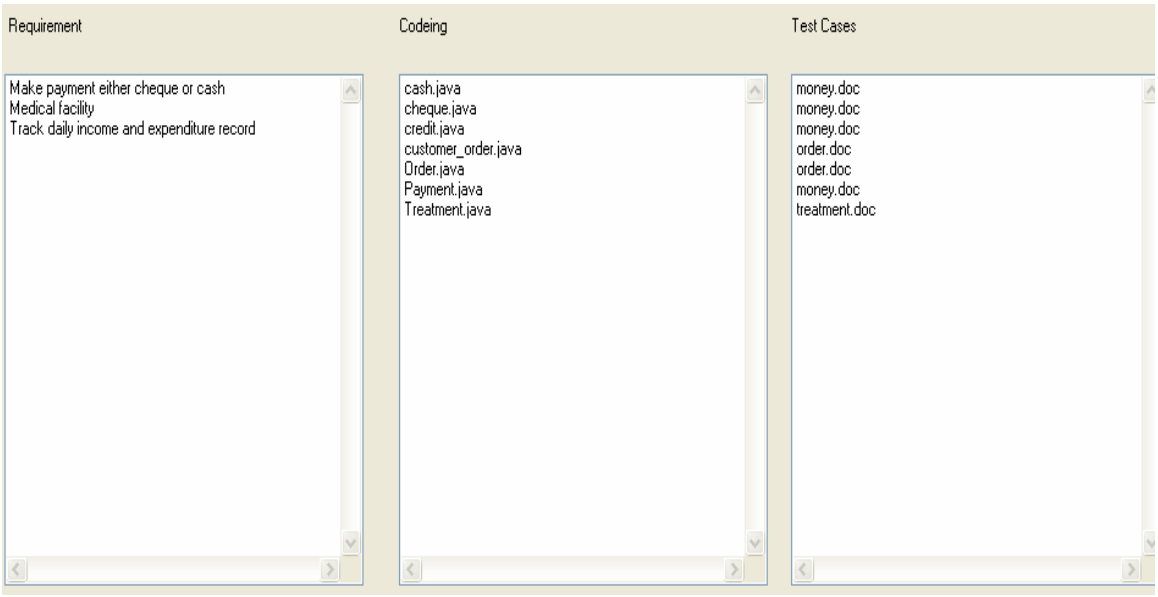
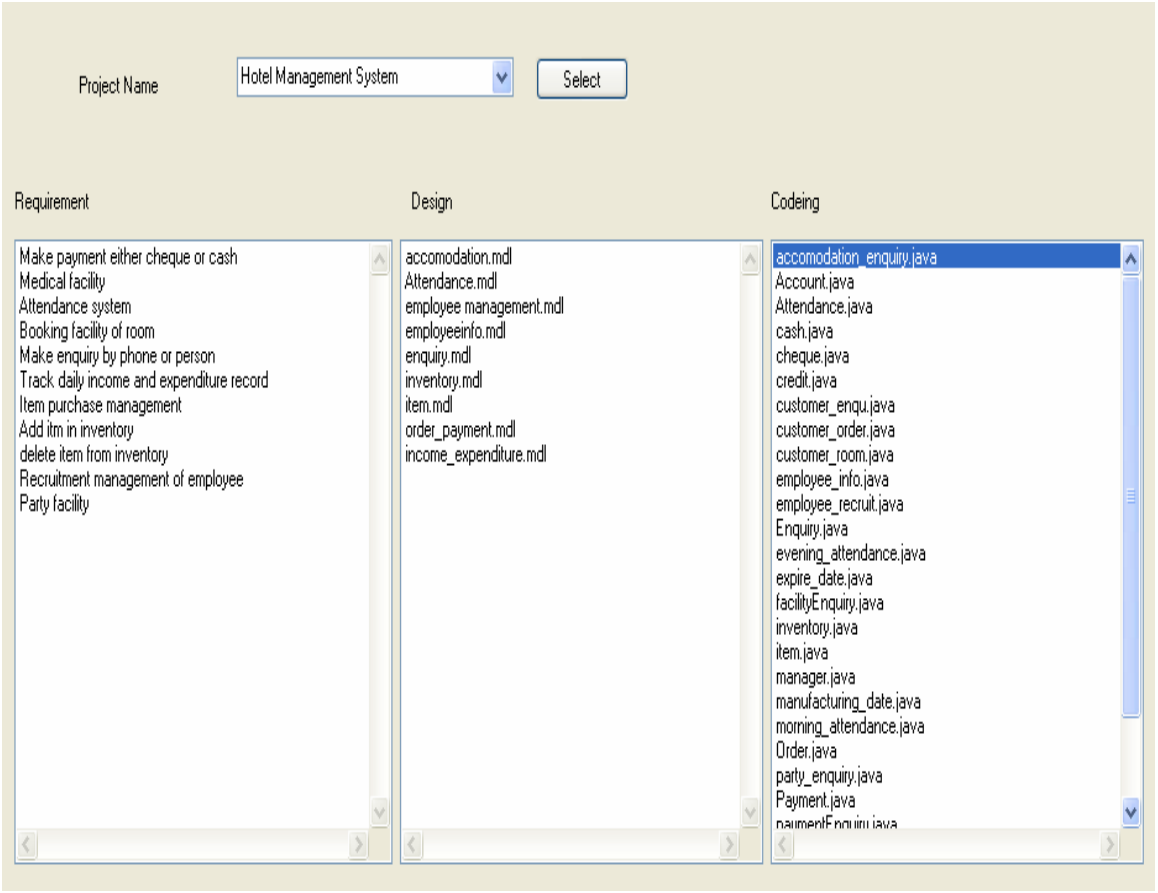


Figure 4.12b: Requirement, coding files and test files affected due to change

4.2.3 Affected Artifacts due to Code Change



4.13a: Selected code file

The user can select any code file whose affect to be found, from codeing window as shown in figure 4.13a.As user clicks the selected code file, the corresponding requirement ,design, code and test files affected are displayed as shown in figure 4.13band 4.13c.

Figure 4.13b: Coding files affected due to change

Figure 4.13c: Requirement, design files and test files affected due to change

Discussion

This has been observed that even a small change in one software artifacts affects all other software artifacts. That further requires a lot of effort to maintain. The proposed tool has been able to identify the change impacts of requirements, design and coding on other artifacts.

Chapter 5

Conclusions and Future Scope

Change in software requirements during software development and maintenance affect design, coding and testing. In the same way change in design and coding affect other software artifacts. The proposed method identifies the change impacts of requirements, design and coding on other software artifacts. The main contribution of this work is that it has been successfully identified the affect of change from requirements development to testing. The major contributions of this work are given below.

5.1 Conclusions

- **Change Impact**

  UML based design is used to explore changes in requirements, coding and test files. It is observed that the results given by tool are as per expectations.

- **Effort Saving**

  As this tool identifies the change impacts automatically, it saves development /maintenance effort to the great extent.

- **Ease in Debugging**

  This tool helps in identifying the test cases of fault and also the impact of this fault on various modules. So it helps to maintain the project in an efficient way.

5.2 Future Scope

- This technique further can be extended to include a module which can automatically make changes in other software artifacts.
- Further work can be explored in the direction of creating a centralized project repository which can record the project change history and its impact analysis.

References

[1] A. Chen, E. Chou, J. Wong, A. Y. Yao, Q. Zhang, S. Zhang, and A. Michail. CVS Search: searching through source code using CVS comments, In Proc International Conference on Software Maintenance (ICSM 2001), pages 364–374.

[2] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases In Proc International Conference on Software Maintenance (ICSM 2000), San Jose, California, USA, IEEE Oct. 2000 pages 120–130.

[3] D.Cubranic and G. Murphy. Hipikat: Recommending pertinent software development artifacts In Proceedings 25th International Conference on Software Engineering (ICSE 2003), New York NY, ACM Press, 2003, pages 408–418.

[4] D. L. Atkins. Version sensitive editing: Change history as a programming tool. In B. Magnusson, editor, Proceedings of System Configuration Management SCM'98, volume 439 of LNCS, 1998, pages146–157.

[5] Grady Booch, James Rum Baugh and Ivar Jacobson. The Unified Modeling language User Guide, 2006 by Pearson education, ISBN 81-7758-372-7.

[6] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In Proc. International Conference on Software Maintenance (ICSM '98), Washington D.C., USA, IEEE Nov. 1998, pages 190–198,

[7] Ian Sommerville. Software engineering ,7th edition,2005, Pearson Education ISBN 81-297-0867-1

[8] J. Bevan, J. E. James Whitehead, S. Kim, and M. Godfrey. Facilitating software evolution research with kenyon. In Proceedings of the 10th European software engineering conference,2005, pages 177–186

[9] J. M. Bieman, A. A. Andrews, and H. J. Yang. Understanding change-proneness in OO software through visualization. In Proc. 11th International Workshop on Program Comprehension, Portland, Oregon, May 2003, pages 44–53.

[10] J. Estublier, D. Leblang, A. van der Hoek, R. Conradi, G. Clemm, W. Tichy, and D. Wiborg-Weber. Impact of software engineering research on the practice of software configuration management. ACM Transactions on Software Engineering and Methodology, 14(4), Oct. 2005, pages 383–430.

[11] Jacek S´ liwerski, Thomas Zimmermann and Andreas Zeller: When Do Changes InduceFixes? MSR'05,Saint Louis, Missouri, USA .Copyright 2005 ACM 1-59593-123-6/ 05/0005, ,May 17, 2005,

[12] K. Fogel and M. O'Neill. cvs2cl.pl: CVS-log-message-to- ChangeLog conversion script, Sept. 2002. http://www.redbean. com/cvs2cl/.

[13] K. A. Schneider, C. Gutwin, R. Penner and D. Paquette. Mining a software developer's local interaction history. In Proceedings 1st International Workshop on Mining Software Repositories (MSR 2004), Los Alamitos CA, IEEE Computer Society Press, 2004,pages 106–110.

[14] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. In Proc. 10th Working Conference on Reverse Engineering (WCRE 2003), Victoria,British Columbia, Canada,IEEE Nov. 2003.

[15] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In Proc. International Conference on Software Maintenance (ICSM 2003), Amsterdam, Netherlands, Sept. 2003. IEEE.

[16] M. Kim and D. Notkin. Program element matching formulti-version program analyses. In MSR '06: Proceedings of the 2006 international workshop on Mining software repositories, 2006, pages 58–64.

[17] MBASE. Avoiding the Software Model-Clash Spiderweb in IEEE Computer, November, 2000, pp. 120-122.

[18] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. InProceedings of the 20th Very Large Data Bases Conference (VLDB), Morgan Kaufmann, 1994, pages 487–499.

[19] R. Conradi and B. Westfechtel. Version models for software configuration management. ACM Computing Surveys, 30(2), June 1998, pages 232–282.

[20] Romain Robbes: Mining a Change Based Software Repository in proceedings of Fourth International Workshop on Mining Software Repositories (MSR'07) ISBN- 0-7695-2950-X/07, 2007

[21] R. Robbes and M. Lanza. Versioning systems for evolutionresearch. In Proceedings of 8th International Workshop on Principles of Software Evolution(IWPSE 2005), IEEE Computer Society, 2005, pages 155– 164.

[22] T. Ball, J.-M. Kim, A. A. Porter, and H. P. Siy. If your version control system could talk, In ICSE Workshop on Process Modelling and Empirical Studies of Software Engineering, 1997

[23] Testing homepage http://www.cs.rit.edu/~afb/20012/cs4/slides/testing-05.html

[24] The Bugzilla Team. The Bugzilla Guide -2.18 Release, Jan. 2005. http:www.bugzilla.org/docs/2.18/html/

[25] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl and Andreas Zeller: Mining Version Histories to Guide Software Changes, IEEE Trans. Software Engg.31(6),2005,pages 232–282.

[26] Thomas Zimmermann, Stephan Diehl and Andreas Zeller: How History Justifies System Architecture (or not), IWPSE 2003, pages 73-83

[27] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In 26th International Conference on Software Engineering (ICSE 2004), Los Alamitos CA, 2004. IEEE Computer Society Press, pages 563–572.

List of Papers Presented

[1] Tarun Kumar Agrawal and Mr. Rajesh Bhatia "Using Knowledge to find impacts in Software" In Proceedings of the "National Conference on Emerging Trends in Engineering and Technology (NCETET)", Murthal, India, May 2008, pp 17-20