CHAPTER 2
SOFTWARE MAINTENANCE USING XP

2.1 INTRODUCTION

Maintenance is an inevitable part in the life cycle of a software product. It accounts for the majority of effort as compared to any other activity of software engineering. Maintenance deals with fault correction, quality improvement and incorporation of new requirements after the product delivery [14]. Maintenance gives an illusion of being an iterative development round of software when observed from exterior [10]. A closer study reveals the differences in the procedures emphasized during maintenance from those applied in the development life cycle of software. During maintenance, more effort is required on the early stages than the development [11] and hence, it is mandatory to have maintenance-conscious models. Different maintenance-conscious models are quick-fix, Boehm, Osborne, iterative-enhancement, full-reuse etc. [11, 12].

The quick-fix approach works on code and allows necessary changes in the associated documentation. Due to time and budget constraints, changes are carried out without proper planning, design, impact analysis, or regression testing, which result in degraded quality of documentation. Degraded documentation and demolished original design add complexity and expense to future modifications [11, 12].
Boehm's model represents the maintenance process as a closed loop [41]. It is based on economic models and principles, which aim to improve productivity with a better understanding of the process. In this model, maintenance starts from management decisions, where management approves a list of changes on the basis of cost-benefit assessments [11]. In Boehm's model, maintenance process is decomposed into four stages; namely, management decisions for
changes, implementation of changes, use of software, and evaluation for the proposed changes.

The Osborne model [42] deals with the realities of the maintenance environment. This model organizes maintenance into different stages, i.e., identification of change requirements, change request submissions, analysis of requirements, change request approval, task schedules, analysis of design, review of design, code modifications, review of changes, test the changes, changes in the documentation, auditing, user acceptance testing, review after the installation, and completion of changes. According to Osborne, lack of proper communication and co-ordination among management are the key reasons behind the technical problems that occur during maintenance. This model recommends a strategy that includes maintenance requirements in the change specification, a software quality assurance program to establish quality assurance requirements, verification of maintenance goals and performance reviews to provide feedback to managers.

In the iterative-enhancement model, the new build creation (i.e., maintenance) begins with the analysis of existing system’s requirements, design, code and test documents [12]. The process proceeds with modifications of the highest-level documents affected by changes, propagating the changes down through the entire documentation. Evolutionary life cycle models are comprised of redesigns based on analysis of the existing system at each step. The full-reuse model is also an evolutionary model. It derives from the reuse-oriented software development [12]. It starts with the analysis of requirements and the design of a new system by reusing the artifacts such as, requirements artifact, design document, source code and testing artifact of the existing system. In this model, a repository of mainly documents and components is used to maintain the artifacts of earlier versions and other similar systems. It promotes the maintenance with reuse of COTS (Commercial Off The Shelf) components.
The IEEE has provided a standard that organizes maintenance process into seven phases, i.e., problem identification, analysis, design, implementation, regression/system test, acceptance test, and delivery [43]. The ISO-12207 standard organizes the maintenance into process implementation, problem and modification analysis, modification implementation, maintenance review/acceptance, migration, and retirement phases [44]. IEEE-1219 and ISO-12207 are based on waterfall-driven traditional methods. IEEE-1219 itself recommends heavy documentation by the use of 41 different IEEE standards for software maintenance [45].

The existing software maintenance models are generally derived from the classic waterfall life cycle and other traditional software development models. The existing maintenance process models have limitations while handling critical problems such as, unstructured code, team morale, poor visibility, and lack of communication among stakeholders. These limitations can be resolved using agile methodologies that focus on customer collaboration, continuous testing, refactoring, and incremental development [18, 45].

Extreme programming (XP) is an implementation of agile philosophy to support evolving nature of customer requirements and quality of software. The main strength of XP is its practices, which are derived from the best practices of software engineering [22, 23]. The twelve practices of XP are categorized into development practices and project management practices. The development practices includes, small releases, metaphor, simple design, tests, refactoring, pair programming, collective ownership, and continuous integration. The project management practices comprises of planning game, 40-hour week, on-site customer, and coding standards.

The core activity of maintenance is code change, which changes the existing code to remove a bug or add new functionality. Maintenance projects often deal with unstructured code due to patched
and repatched software while addressing successive customer issues [18, 30]. The code base of maintenance projects contains bad code. Change in a section of unstructured code without proper test coverage involves risks. Software maintenance processes can be affected by staff turnover, low team morale, poor visibility, complexity of maintenance projects, and lack of communication among stakeholders. The software maintenance process can be bogged down by a lack of proper test suites [20, 31, 32]. On the other hand, TDD is at the heart of XP. TDD provides courage to team members during code change activity. At the same time, refactoring improves the extensibility and readability of the source code with the help of TDD. Other XP practices such as pair programming and collective ownership can mitigate the problems, which arise from staff turnover. Additionally, pair programming can make maintenance tasks more enjoyable, and it encourages the exploration of more alternatives through continuous review.

The literatures show that XP practices applied to software maintenance projects provide better solutions, as discussed in Section 2.2. During the maintenance of non-XP projects, practitioners require a dedicated process model based on XP practices for legacy system maintenance. We have proposed a process model for software maintenance, i.e., iterative maintenance life cycle using extreme programming. It uses Request for Change (RC) stories as requirement artifacts and these are written by the end-users of the software [46]. The process is discussed in the subsequent sections of this chapter. The experimental evaluation and analysis of the proposed process model are discussed in Chapter 6.

The proposed process model incorporates XP practices in different phases and activities of maintenance. The sections of the chapter are organized as follows: Section 2.2 discusses related work on XP practices, which are used in maintenance environments. An illustration of the proposed process model is provided in Section 2.3. Finally, Section 2.4 describes the summary of presented research.
2.2 XP PRACTICES IN MAINTENANCE ENVIRONMENT

XP provides several advantages to the practitioners of software maintenance. It promotes extensive use of unit tests [22, 24]. Extensive test coverage provides several advantages in the maintenance process such as, instant feedback while working on legacy code, confidence and courage while performing error-prone modifications, improved code readability, and faster impact analysis before modifications [18, 47, 48, 49, 50, 51]. It is a prerequisite that the planned changes for performing any refactoring should be supported with test cases. It supports the addition of new features and the fixing of bugs in a safe manner. It is similar to the bottom-up program comprehension approach [23, 52]. However, adding test cases during maintenance of complex legacy code is difficult and also it slows down the maintenance process. Unit tests themselves require maintenance, i.e., extensive unit tests also have to be changed when production code is changed [25]. Before introducing new code in existing legacy code, an automated test suite should be written during the initial iteration [49]. Developing unit tests for a large legacy system all at once is time consuming and, as a practical matter, it is usually infeasible. To solve this problem, some prioritization criteria can be helpful such as divide and conquer on the basis of function size, modification frequency and bug fixing frequency [25].

Refactoring is a set of activities that transforms the design of software system without affecting its existing functionality [53]. Refactoring removes unnecessary complexity, thereby making changes to the code easier and faster [25]. It also improves the structure of the code and hence, it improves readability. In XP, all the tests have to be executed after refactoring [54, 35]. Refactoring can be used in both top-down and bottom-up program comprehension [52]. Refactoring can be applied either by analyzing code for refactoring using bug density analyzed using a bug tracking system or on encountering bad smells in the code [18, 55]. The refactoring process requires courage to make
changes in legacy code for design simplification [27]. Test cases have to be in place for refactoring to be safe. For this reason, tests should be written for the code prior to refactoring [22, 50, 52, 56]. During maintenance, refactoring activities are applied in small steps after writing unit tests [56, 57, 58]. Reverse and forward refactoring activity can be useful for maintenance where, forward refactoring improves efficiency and reverse refactoring improves comprehension quality of the code [59]. With tests, changes can be performed in a better way.

Change operations on legacy code in maintenance projects are often mind-numbing. With pair programming, it might be more enjoyable, cause more alternatives to be explored, and work better on complex problems [25]. Using this practice, knowledge is spread among the maintenance team using frequent partner switching. Pairs provide better results in solving complex problems [28, 29]. They come up with better solutions that are easier to maintain at later stages as decisions are made by the pair. Coding standards are more strictly adhered to improve structure of code and code quality with 15% fewer defects [28, 29]. It is observed that 15% more effort is consumed in pair programming, but XP proponents claim that the time loss is regained because of the improved code quality [28, 29].

The customer collaboration practice of XP provides support for revealing software maintenance needs in an organized way. It provides flexibility to respond rapidly to the changing needs of the customer [45]. Fully automated software testing and continuous integration and build processes reduce the size of source code and maintenance staff by 40%. This makes it easier to maintain, test, and understand the code [18, 30, 60, 61]. The development practices of XP increase maintenance productivity by three times but their adaptations are challenging [31].

The synthesize evolution and maintenance process model, based on subset of agile process shows that the degree of agility is higher in
the implementation phase whereas lower in the pre-implementation phases [62]. Incremental development, risk reduction, and continuous testing contribute to the effective software evolution. The XP practices of metaphor, pair programming, refactoring, small releases, continuous integration with test, continuous testing, and planning games are similar to traditional software maintenance activities of comprehension, debugging, code restructuring, fixes, regression testing, test suites, and change requests [63]. In legacy system maintenance, an XP based methodology increases the speed of delivery, improves user requirements through small releases and customer collaboration. Also, it reduces delivery time, and produces higher quality applications through a test-first approach and code ownership reduces the risk due to staff-turnover [64]. XP provides a right attitude for continuous improvement during maintenance of the system by testing and refactoring [65]. The practice of short iterations is the default for a defect-fixing release during maintenance [32].

During legacy system reengineering, XP makes the process of entire project more visible, controls the progress and cost in a better way, and produces better quality code. The main reason behind the success of XP is the coordination of practices and flexibility [66]. To support the migration of a legacy information system, XP practices and concepts such as planning game, short releases and documentation for need, are beneficial; where the main activities may be design-oriented [67]. During legacy system reengineering, iterative developments can reduce the impact of communication delay [68]. During the evolution of non-XP legacy code, productivity is lower than regular XP, but testing and continuous integration provide important instant feedback. XP does not concentrate on designing architecture and documentation as these are mostly required during long term maintenance and evolution of software [25].

Each of these practices of XP has their own advantages and limitations. There does not exist any standard process model for
software maintenance that could integrate all XP practices. Therefore, a maintenance process model is proposed covering the entire set of XP practices. It is iterative in nature. The main phases of this process model are identification and categorization, planning, analysis, design revision, change implementation, acceptance testing, and release. This process model applies changes in an iterative manner. The proposed model with its phases is discussed in subsequent sections. The proposed approach includes the benefits of existing practices and provides a standard model for maintenance. Besides that, it is an XP based approach and it includes documentation for future maintenance activity.

2.3 ITERATIVE MAINTENANCE LIFE CYCLE USING XP

There exist several approaches and models for software maintenance, which are derived from traditional software development process models. We have proposed an iterative maintenance life cycle using extreme programming, as shown in Figure 2.1. The IEEE 1219 [43] is considered as a framework in the design of proposed process model. The phases and activities of IEEE 1219 are adapted and extended in the proposed process model from an XP perspective. It consists of seven phases; namely, identification and categorization, planning, analysis, design revision, change implementation, acceptance testing, and release.

The proposed approach uses RC stories and old software as input and performs all the phases and produces a modified product inheriting quality attributes such as, improved maintainability, increased productivity of maintenance team, reduced cost and effort of software maintenance. The specifications for changes or enhancements in the existing functionality or reported bugs are considered in the form of RC stories. The individual phases of proposed process model along with an example of a Student Feedback System (SFS) are discussed in the following subsections.
A maintenance project, SFS, is considered to explain the different phases of proposed maintenance process model. It is a web based application for an educational institution developed by earlier group of students by applying the ad-hoc techniques. Students can register and provide feedback about an instructor through a questionnaire. After successful submission of feedback, college management can view the feedback in different formats and can generate reports for various purposes. The RC stories which are requested for maintenance are listed in Table 2.1.
Table 2.1: RC Stories for SFS

<table>
<thead>
<tr>
<th>Story ID</th>
<th>Description</th>
<th>Type of Maintenance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RC1</td>
<td>Add attendance criteria as an eligibility for feedback submission</td>
<td>Corrective</td>
</tr>
<tr>
<td>RC2</td>
<td>College management can view and print feedback report department wise</td>
<td>Perfective</td>
</tr>
<tr>
<td>RC3</td>
<td>Provide suitable faculty for a subject</td>
<td>Perfective</td>
</tr>
<tr>
<td>RC4</td>
<td>Add a field of the number of lectures conducted in feedback form</td>
<td>Perfective</td>
</tr>
<tr>
<td>RC5</td>
<td>Administrator can dynamically change feedback question</td>
<td>Perfective</td>
</tr>
</tbody>
</table>

2.3.1 Identification and Categorization

The identification and categorization of change requests is performed at this phase. It deals with seeking change requests in the form of RC stories as an input and produces validated and categorized RC stories as output, as shown in Figure 2.2. The RC story is a requirement artifact, which is written and submitted by end-users [46]. The RC story has two parts: a description of the change requirement and user acceptance criteria. The requirements description of an RC story consists of a problem statement and its priority. After the submission of an RC story, a unique identifier, Story ID, is assigned, which can be used to track the status of the RC story. Skilled people such as system analysts or service engineers examine an RC story for reasonableness with the support of an on-site customer. RC stories are assigned one of the maintenance categories such as, corrective, perfective, or adaptive maintenance. The identification and categorization phase is performed under planning game practice.
In the case of SFS maintenance project, RC stories submitted by college management are categorized into one of the maintenance categories. RC1 belongs to the corrective and the other four RC stories are perfective. Using on-site customer practice, the end-users of SFS provide explanations for the RC story and support during validation of RC stories.

2.3.2 Planning

The planning phase follows the planning game practice of XP, which mainly focuses on the development of strategies such as, release plan, and iteration plan for the completion of RC stories. Inputs for this phase are RC stories along with the priority and the type of maintenance, as shown in Figure 2.3. Developers along with customers make decisions about the grouping of RC stories into patch release iterations. Generally, RC stories, which belong to the same category of maintenance, are developed in a single release. If the maintenance category is urgent repair then the request is implemented irrespective of analysis and design.

There are certain issues during a planning game in the case of maintenance; it is difficult to decide about the priority of implementation of new features and improvements of code design. This problem can be solved by treating the initial iteration as corrective maintenance. The corrective maintenance can improve the design of existing legacy code. Bug cluster information in a bug tracking system can support this structural improvement activity by detecting bad smells. Another issue is the inaccurate estimation of efforts caused by unfamiliarity with the existing code. The planning poker technique can be beneficial for estimation at this stage of maintenance life cycle as compared to the estimation performed by an individual. During release planning, the release date is also decided based on the estimation. For a single release, cost analysis and estimation are performed to estimates the response time for completing an RC story.
Figure 2.2: Identification and Categorization Phase

RC Stories → Identification and Categorization → Validated and Categorized RC Stories

On-Site Customer → Planning Game

Figure 2.3: Planning Phase

Validation and Categorized RC Stories → Planning → Iterative Planning

On-Site Customer → Planning Game → Small Releases

Release plan and Iteration Plan
The agreement between developer and customer is signed based on included stories and agreed upon release date. During iteration planning, RC stories are divided into tasks, which are assigned to the programmers.

In case of SFS maintenance, RC1 will be implemented in the first release because it is corrective maintenance and the customer has assigned a higher priority to it. The second release considers the other four stories, which belong to the same category of maintenance. The project coordinator prepares a panel along with students and project guides that follows planning poker techniques. RC1 is estimated as eight story points.

2.3.3 Analysis

The analysis phase aims to determine the feasibility of maintenance for a project by identifying the impact of modification, using an analysis of alternative solutions and effort estimation. The release and iteration plan along with artifacts of the existing system are the main inputs to this phase. Outputs of the analysis phase are the feasibility and detailed analysis reports, as shown in Figure 2.4. The effort involved in this phase is highly dependent on the quality and availability of the artifacts of the system [69]. Analysis is conducted at the following two levels [43]:

- Feasibility analysis
- Detailed analysis

During feasibility analysis, feasibility and scope of modifications are determined. It is comprised of the following three activities [43]:

- Identification of the impact of modifications
- Analysis of alternative solutions
- Identification of safety and security implications
The first activity during feasibility analysis is to identify the impact of modifications. The purpose of this activity is to evaluate the scope and impact of modifications and then to decide whether or not to proceed with implementing an RC story. A valid RC story and its relevant artifacts serve as inputs to this activity. The impact of modifications is analyzed roughly based on existing documents, specifications and source code with the purpose of identifying affected products, artifacts and modules that need to be modified. The impact of modification on the business process with short and long-term costs is also evaluated. Visual modeling tools are useful to demonstrate the scope and tasks of the system. In case of SFS, the project guide along with the students perform this activity. The purpose is to identify the impact of RC1 from product and end-users points of view.

The other activity performed during feasibility analysis is the analysis of alternative solutions. It identifies and assesses the different alternatives for a given RC story. The RC story with its relevant artifacts becomes an input for this activity. During assessment,
prototyping techniques can be useful in evaluating alternatives. The best alternative is selected amongst the possible solutions, which is implemented at a later stage. The on-site customer can supply support for identification and selection among alternative solutions. Requirements elicitation is performed by the analyst. The software architect along with the designer performs the design and analysis of alternatives. The project manager and on-site customer approve the recommended solution. Prototyping and simulation tools can be helpful for the analysis of alternative solutions [69].

For RC1, several business alternatives are considered as the means to include percentage criteria in SFS. The first alternative is to allow only administrator to fill the attendance of each student. Students with below average attendance will be notified that they are not eligible to provide feedback through a message. The second alternative is to allow each student to fill his/ her attendance and on the basis of minimum attendance criteria he/ she will be eligible to fill feedback for that subject.

The identification of safety and security implications is the last activity of feasibility analysis phase. The purpose of this activity is to identify all other aspects that can compromise success with regard to safety and security. Input for this activity is an RC story with detailed information about the system. Current safety and security issues along with the data security policy of the organizations are also used as inputs for this activity. Firstly, identification of vulnerable areas of the system is performed and their successful identification is followed by analyzing the RC story.

The detailed specification of safety and security implications is performed in later stages of the process model. Output of this activity is a list of the aspects that should be considered in detail during later stages, i.e., detailed analysis, design, and testing. Identification of safety and security areas is performed by the system analyst. In the
future, the system analyst, architect and designer specify the implications in more detail [69]. An automated tool (e.g., Eclipse IDE) to trace the functionality in the existing code is required during detailed analysis of safety and security implications.

According to the rule, the faculty members are not allowed to view their feedback; it is a confidential part of SFS. For RC1, the project guides along with the coordinator perform identification and analysis activities from safety and security points of view.

On a positive feasibility report, a thorough analysis is required for design, which is performed during detailed analysis. It is comprised of the following four activities [43]:

- Define firm requirements for the modification
- Identify the elements of modification/impact analysis
- Program comprehension
- Effort estimation

First activity of the detailed analysis phase defines firm requirements for the modification. It establishes and maintains agreement of what the system should do after implementation of the RC story. It determines the scope of an RC story that provides a basis for the design. The RC story, analysis report and original Software Requirement Specification (SRS) are inputs for this activity. In case of corrective maintenance, the problem can be reproduced to understand the requirements. Use case diagrams can be used to represent the user requirements. This activity results in the analysis model, which represent the system or its affected units. The requirement definition activity is coordinated by the system analyst. The analysis model is developed by the designer and the integrity of the existing system architecture is ensured by the architect [69]. The RC story and its traceability can be captured by an automated requirement management
tool. Analysis modeling tools provide different views of the system during the analysis process.

As the first alternative is selected in feasibility analysis of the RC1 in SFS, firm requirements are established for the modification. According to the first alternative, from the administrator side, the faculty member of a related subject has filled the name, enrollment and attendance of the student for eligibility. A student can provide feedback, if he/she have an attendance greater than 75%. A student will receive a message of eligibility during subject selection.

Second activity during detailed analysis is to identify the modification and analyze their impact. This activity assesses the impact of modifications in the system, which provides a basis for the design. Inputs for this activity are an RC story, analysis model, and relevant artifacts. The impact of modification on other components of the system is identified in this activity. This activity produces improved analysis models that provide a general idea about intended functionality, elements that need to be modified and the impact of changes.

The impact analysis is performed for RC1 in the SFS. The affected modules are identified on the basis of firm requirements. During this activity, analysis models such as use case and class diagram are modified according to the firm requirements of RC1.

The program comprehension is the third activity of detailed analysis, which is used to understand the existing source code. A program analyzer or program slicing technique can be used to understand the code [69]. A detailed document is produced that assists others to understand the system’s functionality and behavior during modifications. The pair programming practice can be used for the program comprehension task. The entire technical team is involved in
this activity according to their roles. Code review and dynamic analyzer tool support the activity of program comprehension.

The last activity of the detailed analysis phase is effort estimation. The effort estimation of a maintenance project is highly influenced by the existing design documents and source codes. Cost drivers such as product, computer, personnel, and project play important roles during estimation [70]. The SMEEM model can be applied to calculate the volume of maintenance [71, 72]. The SMEEM of effort estimation incorporates value adjustment factors such as, documentation quality, structuredness etc., with different intensity levels. This model helps project managers to calculate the estimated software maintenance effort in terms of Adjusted Story Point (ASP), size, cost and duration.

We apply SMEEM on SFS to calculate unadjusted effort \( (E_{\text{Unadjusted}}) \), which is 16 story points. The value of different value adjustment factors for Factor Count (FC) is computed as 24. Effort is computed as 7.5 hours for RC1 by applying SMEEM.

This phase considers the impact of RC stories on rest of the modules with consideration to their security and safety implications as well. The analysis phase concludes with estimation of cost, size and duration.

**2.3.4 Design Revision**

In this phase, changes are performed on the existing design artifacts without affecting the integrity of existing system. Feasibility and detailed analysis reports are the main inputs to this phase with program comprehension as an important prerequisite. The output of the design revision phase is the updated design baseline, as shown in Figure 2.5. This phase is comprised of the following three activities [43]:

- Modify software module documentation
- Create test cases for the new design
- Identification and creation of integration tests

The documentation of software modules is modified initially during the design revision phase. Inputs to this activity are an RC story and relevant affected modules. The purpose of this activity is to convert RC stories into specification documents represented in the form of modules. The specification module is represented using CRC (Class Responsibility Collaborator) cards. This design activity uses refactoring and metaphor practices to keep the system design simple and extensible. Outputs of this activity are design elements such as, class diagram and subsystem design, which represents different views of the system. In this activity, an architect provides the architectural design of the overall software and the designer provides the design modules. In this activity, the Rational Software Architect (RSA) tool performs reverse and forward engineering.

![Diagram showing the Design Revision Phase]

**Figure 2.5: Design Revision Phase**
The second activity during the design revision phase is to create test cases for the new design including safety and security issues. The purpose of testing at this stage is to develop a test model that can be used during verification and validation of an RC story. Input for this activity consists of existing testing artifacts and revised design artifacts from the first activity. In this process, an RC story and design models are converted into acceptance and system test cases, respectively. Outputs of this activity are test plan, test cases, test procedures, and test scripts. The test designer performs planning, designing, implementation, and evaluation of the tests. During this activity, automated tools can support the creation and execution of test cases. Modeling tools can also support conversion of use cases into test cases and test scripts.

Planning for integration testing is the last activity of the design revision phase. At this stage, it is easier to identify affected areas that are beneficial for planning of integration testing. Defects, which are discovered earlier, existing test cases and design information from the design model are the inputs for this activity. The test plan is established according to the modified system and then test cases are designed and implemented accordingly. Test plan, test cases, and test script are the outputs of this activity. During continuous integration testing, it is necessary to re-run test cases again and again; therefore, automated tools can be useful for efficient execution of test cases. Overall, this phase incorporates changes into the design of existing system. It includes creating test cases for new designs and planning of the integration testing.

2.3.5 Change Implementation

Coding and unit along with integration testing are performed in the change implementation phase. It takes updated design documents and existing source codes including production and test codes as input and produces updated software as an output. Pair programming, TDD,
refactoring, and continuous integration practices of XP can be used during this phase, as shown in Figure 2.6. The change implementation phase is comprised of the following three activities [43]:

- Structure implementation
- Test-first coding
- Continuous integration and test

In case of urgent repair, the modification of code can be performed as an immediate effect. After performing code changes, changes in the design document are applied strictly. During this phase, an RC story may be divided into multiple tasks.

Initially, a structure for the change implementation is prepared that represents the structure of code change along with the consistency. The detailed design documents are inputs to this activity. It starts from the analysis phase, where modified components are already identified. Here, an implementation strategy is also prepared that describes a sequence of modifications.

![Figure 2.6: Change Implementation Phase](image-url)
Planning the integration of modified module is also performed at this stage. Implementation models and integration plans are outcomes of this activity. Software architects play important roles during the preparation of the implementation model and integration plan.

Performing changes in the code is the core activity of this phase. It aims to develop components using design objects. The design model, implementation model, existing source code, and existing documentation are inputs for this activity. This activity can be performed by the code change approach [73], which employs the TDD, pair programming and refactoring practices of XP. Firstly, unit test classes are written or existing test classes are modified according to the requested changes. In case of test class unavailability, unit test classes are written according to the structure of actual class. The program comprehension can be applied using pair programming while writing unit tests. In the initial iterations, emphasis should be given on writing unit test classes to obtain more and more coverage for existing code that helps the design of actual classes. Test and actual class creation or modification is performed using the pair programming practice. The refactoring practice can also be applied in small steps to incorporate code changes. Outputs of this activity are unit test classes along with actual classes. Automated tools that support unit testing along with refactoring such as eclipse can increase the productivity. The round-trip engineering tools are also helpful here. During this activity, a sustainable pace is maintained with the help of the 40-hour week practice of XP. The collective code ownership practice can also be followed here so that every programmer can be aware of changes in different modules of the system. The daily stand-up meeting practice can also be followed for briefing regarding the tasks for the day, technical issues and for information sharing among the team members.

Lastly, continuous integration and testing is performed in the change implementation phase. The purpose of this activity is to integrate the work of individual programmers into an executable unit.
The integration plan, existing code, and tested units are inputs for this activity. Continuous integration along with testing is performed on a daily basis. The continuous integration and testing practice of XP can resolve compatibility issues and interfacing problems of maintenance. This practice during maintenance provides a smoke-testing environment to detect errors in early stages. An integrated system or subsystem, which is ready for acceptance testing, is the output of this activity. An integration team or pair programmer performs this activity. Automated tools such as, TeamCity, Bamboo, and CruiseControl can provide support during integration and management.

### 2.3.6 Acceptance Testing

The RC story driven acceptance testing is performed at this stage to ensure that the components and the system as a whole provides expected results. The acceptance test scripting activity starts from the analysis phase for all those RC stories that are planned for the implementation. Integrated modules and testing artifacts are the inputs to this phase, as shown in Figure 2.7. The acceptance testing phase is comprised of following three activities [43]:

- Test design
- Productive system test
- Support material acceptance test

![Figure 2.7: Acceptance Testing Phase](image-url)
Initially, test design is performed for an RC story. The RC story as an input to this activity contains acceptance criteria for requirement change. A user interface designer creates a prototype by means of an RC story acceptance criteria to design the tests. A test designer designs the test script on the basis of customer feedback. User approved test cases for acceptance testing is the outcome of this activity. The test designer implements test scripts, and the customer provides feedback, which acts as a source for the improvement.

Productive system testing is the core activity of this phase, which aims to ensure that the operational system is acceptable for the end-users. The operational system, test cases, and test scripts, which are developed during early phases, are used as inputs for this activity. The modified system is tested at the developer’s site and in the operational environment by the end-users. The productive system is the outcome of this activity. The on-site customer along with the tester execute test scripts to check the functionality. Automated testing tools can be used to speed-up test case execution activity.

The support material acceptance testing is the last activity towards release of the system. User manuals and training materials are reviewed at this stage. The entire support material including installation guide, user manuals, training materials etc., serve as input. The tester along with the on-site customer performs the installations and operations based on user manuals and supporting materials. On the basis of the feedback, supporting material is updated before deployment of the system. Acceptable and appropriate support materials are the outcome of this activity.

2.3.7 Release

The release phase accepts tested software as an input and produces customer feedback in the report, as shown in Figure 2.8. This phase comprised of the following five activities [43]:
• Installation of the software at the customer site
• Final testing
• Notification of users/ customers
• Deployment
• User training

Initially, the modified system is installed in the productive environment at the customer site. The ready product along with information about productive environment are the main inputs for this activity. A back-up of old version of software is also maintained for tracing of undiscovered faults. Data migration from the old version to the modified version is also performed at this stage. The installed software in the user’s environment is the outcome of this activity. The deployment manager and the implemeneter perform this activity with the support of users. Back-up and data migration tools can accelerate this activity.

Figure 2.8: Release Phase

The final testing is performed after successful installation of the modified software at the customer site. Approval from the customer is also collected at this stage. The software product, test data and a copy of the agreement are the key inputs for this activity. During the final testing, developers test the product at the customer site, thereafter the
customer tests the same product to ensure that the contract is realized. The functionality of final product can be verified by end-users through parallel working on the old and modified versions of the software with sets of similar data. At the end of a calendar day, a report is generated to compare the functionality of the old and modified versions of the system. This practice provides confidence towards the use of the modified system. Test results along with a running system are the outcomes of this activity. The test designer, tester and end-user perform final testing of the system.

After the successful final testing, a notification is provided to users/ customers about the modification. The list of modifications and affected users/ customers become the inputs for this activity. The affected customers are identified and informed about modifications. The notifier along with the notified users are the outcomes of this activity. The project manager is responsible for the notification task. Communication channels as well as database tools, which provide information about the affected users, are important for this activity.

Through the deployment activity, the modified system is made available to its users for use. Information about affected users and customers along with installation artifacts are inputs to this activity. The deployment activity requires higher customer collaboration and preparation. The on-site customer and planning game practices of XP can motivate users and customers during the deployment activity. The revised system is updated with customer data by providing a link according to the deployment plan with the updated system as an outcome. The deployment managers deploy the software at the customer site on the basis of feedback from customer’s acceptance and test results.

Finally, training sessions are provided to users based on their roles in the organization, viz. installation, operation, maintains, and use of the system. The actual or test system acts as an input in training
activity. To produce trained users, information about the system, modified areas and affected processes are desirable during training activity. Training sessions are highly influenced by the acceptance testing activity of preceding phase, which is performed to test the support material of the system. The person who prepares training material and the trainer play key roles during this training activity. Automated tools such as e-learning can support training activities.

All the phases and activities of each phase are discussed here to illustrate the proposed process model. The experiment and analysis related to the model are illustrated in the Chapter 6.

2.4 SUMMARY

Software maintenance is the continuous process of enhancing the operational life of software. Maintenance of legacy systems is tedious, expensive, and error prone, due to fractional test coverage, inaccessible original developers and insufficient or outdated documentation. The existing approaches to software maintenance derived from the traditional approaches to development are unable to resolve the problems of unstructured code, team morale, poor visibility of the project, lack of communication, staff turnover, and lack of proper test suites. Alternatively, literature shows that XP practices applied to software maintenance projects provide better solutions. XP practices such as, test driven development, refactoring, pair programming, continuous integration, small releases, and collective ownership help to resolve the aforesaid problems. During the maintenance of non-XP projects, practitioners require a dedicated process model based on XP practices for legacy system maintenance.

In this chapter, iterative maintenance life cycle using extreme programming, a process model for software maintenance is proposed to resolve maintenance issues in an improved manner. The proposed approach uses RC stories and old software as input and performs all the
phases and produces a modified product. It reveals the benefits of integrated XP practices in different activities of maintenance. The proposed approach speeds up the maintenance process and produces higher quality code with less effort for future maintenance and evolution. The proposed model based on XP enhances both learning and productivity of the team by improving the morale, courage, and confidence of the team, which supports higher motivation during maintenance. The proposed process model has been applied and validated by using academic case studies, which is discussed in Chapter 6.