A Computer-Based Holistic Approach to Managing Progress of Distributed Agile Teams

Sultan Abdulaziz Alyahya

School of Computer Science & Informatics
Cardiff University

This thesis is submitted in partial fulfilment of the requirement for the degree of Doctor of Philosophy

May 2013

DECLARATION

This work has not been submitted in substance for any other degree or award at this or any other university or place of learning, nor is being submitted concurrently in candidature for any degree or other award.

Signed ………………………………………… (candidate)
Date …………………………

STATEMENT 1

This thesis is being submitted in partial fulfillment of the requirements for the degree of PhD.

Signed ………………………………………… (candidate)
Date …………………………

STATEMENT 2

This thesis is the result of my own independent work/investigation, except where otherwise stated. Other sources are acknowledged by explicit references. The views expressed are my own.

Signed ………………………………………… (candidate)
Date …………………………

STATEMENT 3

I hereby give consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed ………………………………………… (candidate)
Date …………………………

Abstract

One of the co-ordination difficulties of remote agile teamwork is managing the progress of development. Several technical factors affect agile development progress; hence, their impact on progress needs to be explicitly identified and co-ordinated. These factors include source code versioning, unit testing (UT), acceptance testing (AT), continuous integration (CI), and releasing. These factors play a role in determining whether software produced for a user story (i.e. feature or use case) is ‘working software’ (i.e. the user story is complete) or not. One of the principles introduced by the Agile Manifesto is that working software is the primary measure of progress.

In distributed agile teams, informal methods, such as video-conference meetings, can be used to raise the awareness of how the technical factors affect development progress. However, with infrequent communications, it is difficult to understand how the work of one team member at one site influences the work progress of another team member at a different site.

Furthermore, formal methods, such as agile project management tools are widely used to support managing progress of distributed agile projects. However, these tools rely on team members’ perceptions in understanding change in progress. Identifying and co-ordinating the impact of technical factors on development progress are not considered.

This thesis supports the effective management of progress by providing a computer-based holistic approach to managing development progress that aims to explicitly identify and co-ordinate the effects of the various technical factors on progress. The holistic approach requires analysis of how the technical factors cause change in progress. With each progress change event, the co-ordination support necessary to manage the event has been explicitly identified.

The holistic approach also requires designing computer-based mechanisms that take into consideration the impact of technical factors on progress. A progress tracking system has been designed that keeps track of the impact of the technical factors by placing them under control of the tracking system. This has been achieved by integrating the versioning functionality into the progress tracking system and linking the UT tool, AT tool and CI tool with the progress tracking system.

The approach has been evaluated through practical scenarios and has validated these through a research prototype. The result shows that the holistic approach is achievable and helps raise awareness of distributed agile teams regarding the change in the progress, as soon as it occurs. It overcomes the limitations of the informal and formal methods. Team members will no longer need to spend time determining how their change will impact the work of the other team members so that they can notify the affected members regarding the change. They will be provided with a system that helps them achieve this as they carry out their technical activities. In addition, they will not rely on static information about progress registered in a progress tracking system, but will be updated continuously with relevant information about progress changes occurring to their work.

Acknowledgements

Above all, I praise Allah (God) for his uncountable favours and for providing me with patience, strength and persistence to complete this work.

I would like to express my sincere gratitude and thanks to my first supervisor Dr. Wendy Ivins for her unlimited support, patience and encouragement during the period of my study. She was always eager to listen and discuss my ideas, findings and even my concerns. In addition, I would like to thank my second supervisor, Prof. Alex Gray, for his advice and expert guidance during important stages in my research.

I am grateful for the feedback I received from many people in the agile community at XP 2008, ICGSE 2011 and SERA 2012 conferences. Their valuable comments and discussions during the conference days have added to the success of this work.

I would like to thank the members at the School of Computer Science & Informatics at Cardiff University for their help, especially Helen Williams and Dr. Pamela Munn for their help with administrative issues, and Kirsty Hall for her help with travel related issues.

I acknowledge, with grateful thanks, my employer, King Saud University (Riyadh, Saudi Arabia), for granting my scholarship, and providing the financial support needed to complete my study.

I wish to thank my colleagues, Fahad Al-Wasil, Badr Aldaihani, Ahmed Alqaoud, Waleed Alnuwaiser, Yaser Alosefer, Yahya Ibrahim, Ahmed Alazzawi, Ehab ElGindy, Abdul Hamid Elwaer, Khaled Almuzaini and Abdolbast Greede for their friendship and help. Very special thanks are due for Hmood Al-Dossari for the long hours he has spent with me discussing this research, which is absolutely unrelated to his own.

Special admiration and gratitude must go to my parents, brothers and sisters whose prayers, love, patience, support and encouragement have always enabled me to perform to the best of my abilities.

Last, but by no means least, I cannot find words to express my gratitude to my wife, Muneera, who stood by me during the hardest times of the project and provided me with infinite love, care and confidence. Finally, I am greatly indebted to my son, Abdulaziz, for putting a smile on my face when I have needed it the most.

Dedication

To the Egyptian thinker & philosopher Dr. Mustafa Mahmoud (1921-2009) who taught me that the value of human is what he adds to life between his birth and death.

Acronyms

APM	Agile Project Management
ASD	Adaptive Software Development
AT	Acceptance Testing
ATs	Acceptance Tests
CFD	Cumulative Flow Diagrams
CI	Continuous Integration
CSCW	Computer-Supported Co-operative Work
DFD	Data Flow Diagrams
DSDM	Dynamic Systems Development Method
FDD	Feature-Driven Development
IMs	Instant Messaging
IV	Integrated Version
RERO	Releasing Early and Releasing Often
RTF	Running Tested Features
RUP	Rational Unified Process
RV	Releasable Version
SBE	Scenario-Based Evaluation
SCM	Software Configuration Management
TDD	Test-Driven Development
TFS	Team Foundation Server
TTM	Time-to-Market
TV	Transient Version
UML	Unified Modeling Language
UT	Unit Testing
UTV	Unit-Tested Version
WIP	Work In Progress
XP	Extreme Programming

Contents

Declarations/Statements .... **i**

Abstract .... **ii**

Acknowledgements .... **iii**

Dedication .... **iv**

Acronyms .... **v**

1. Introduction **1**

1.1 Problem Statement ***1***

1.2 Hypothesis, Aims and Objectives ***3***

1.3 Achievements of the Research ***5***

1.4 Structure of the Thesis ***7***

2. Agile Software Development **9**

2.1 Limitations of the Plan-Driven Approach ***9***

2.2 Agile Approach ***12***

2.2.1 Fundamentals of Agile Software Development 13

2.2.2 The Concept of Progress in Agile Approach 14

2.2.3 Extreme Programming 15

2.2.3.1 XP Process and Terminology *16*

2.2.3.2 XP Values and Practices *17*

2.3 Technical Factors Affecting Agile Development Progress ***20***

2.3.1 The Technical Factors 20

2.3.2 The Technical Factors in Agile Methods 25

2.4 Summary ***26***

3. Managing Development Progress in Distributed Agile Projects **28**

3.1 Managing Progress of Agile Software Development ***28***

3.2 Distributed Software Development ***30***

3.2.1 Motivations for Distributing Software Development 30

3.2.2 The Need for Co-ordination Support 32

3.3 Current Approaches to Managing Progress in Distributed Agile Projects ***33***

3.3.1 Informal Methods 34

3.3.1.1 Synchronous Communication *34*

3.3.1.2 Asynchronous Communication *36*

3.3.1.3 Daily Tracker *37*

3.3.1.4 Information Radiators *37*

3.3.1.5 Cross-location Visits ... *38*
3.3.2 Formal Methods ... 38
3.3.2.1 Wikis and Spreadsheets ... *38*
3.3.2.2 Traditional Project Management Tools ... *39*
3.3.2.3 Agile Project Management (APM) Tools ... *40*
3.3.2.3.1 Web-Based Task-Board ... 42
3.3.2.3.2 Progress Reporting ... 43
3.3.2.3.3 Time Tracking ... 44
3.3.2.3.4 AT Progress Tracking ... 44
3.3.2.3.5 Progress Notifications ... 45
3.4 Justification for Computer-Based Holistic Approach ... ***47***
3.5 Summary ... ***50***

4. Co-ordination Support Required for Managing Progress of Distributed Agile Projects ... **52**

4.1 Understanding Co-ordination ... ***52***
4.2 Types of Co-ordination Activities Required for Managing Development Progress ***56***
4.3 Analysis of Co-ordination Requirements for the Technical Activities ... ***60***
4.3.1 Source Code Versioning ... 60
4.3.2 Continuous Integration and Releasing ... 63
4.3.3 Unit Testing ... 65
4.3.4 Acceptance Testing ... 66
4.4 Examples ... ***69***
4.5 Summary ... ***72***

5. Design of Progress Tracking System ... **74**

5.1 System Architecture ... ***74***
5.2 Version Model ... ***76***
5.2.1 Version States ... 76
5.2.2 Version Operations ... 77
5.2.3 Version Tracking ... 79
5.3 User Story Progress Model ... ***81***
5.4 Process Model ... ***84***
5.4.1 Selecting a Process Modelling Technique ... 84
5.4.2 Modelling the Technical Processes ... 85
5.5 Data Model ... ***89***
5.6 Design Issues ... ***92***
5.6.1 Acceptance Testing Approaches ... 92
5.6.2 Continuous Integration Approaches ... 93

5.7 Summary 97

6. Evaluation 99

6.1 Evaluation Methodology 100

6.1.1 Evaluation for Groupware Systems 100

6.1.2 A Scenario-Based Evaluation Approach 102

6.2 Selection of Scenarios 103

6.3 Analysis of Scenarios 107

6.3.1 Scenario 1: 'Check-in Source Code Version' Scenario 108

6.3.2 Scenario 2: Performing Successful Integration 114

6.3.3 Scenario 3: Running Automated Acceptance Testing 120

6.4 Validation of the Holistic Approach 125

6.4.1 System Database 126

6.4.2 Implementation of Scenario 1 127

6.4.3 Validation Discussion 133

6.5 Further Discussion on Evaluation 134

6.5.1 Overcoming the Limitations of the Informal Methods 134

6.5.2 Overcoming the Limitations of the Formal Methods 135

6.6 Summary 136

7. Conclusions 137

7.1 Achievement of the Research Objectives 137

7.2 Future Work 146

7.2.1 Impact of Progress Change on Overall Project Plans and Velocity 146

7.2.2 The Use of Change Impact Analysis Techniques 147

7.3 Contribution of the Research 148

7.3.1 Research Publications 148

7.3.2 Originality of the Proposed Approach 149

Appendix A: Agile Principles 151

Appendix B: Technical Process Models 152

Appendix C: Implementation Description for the Holistic Approach Version of Scenarios 2 and 3 167

Bibliography 180

CHAPTER 1

Introduction

This chapter presents an outline of the research. It describes the problem statement of the research in Section 1.1. The scope of the research is defined in Section 1.2 by describing the hypothesis, aims and objectives of the research. The achievements of the research are discussed in Section 1.3. Finally, the structure of the thesis is presented in Section 1.4.

1.1 Problem Statement

The practice of distributed software development has rapidly increased over the last two decades [1] [2]. In spite of applying development methods, co-ordination is one of the primary challenges in developing software at multiple sites [3]. The temporal, geographical and socio-cultural barriers impose a co-ordination challenge to the distributed teams [3] [4].

Distributed software projects using agile processes are likely to encounter more complex co-ordination problems, because agile processes were originally aimed at single location projects, where teams rely on intensive communications among team members to co-ordinate their work. However, an increasing number of agile organisations work remotely to gain the advantages of distributing the work [5] (e.g. the promise of round-the-clock development). With the absence of face-to-face interactions, numerous co-ordination difficulties are reported (e.g. [6–8]).

One of the co-ordination difficulties of remote agile teamwork is managing the progress of development. Sauer [9] points out that progress status is less visible and controllable in distributed agile projects. Peng [10] observes that *"teams have a difficult time keeping track of progress"* in a distributed agile project. Agile software teams may reach the end of a development iteration having a large number of failed acceptance tests, delivering progress information late, and to the wrong team members [11].

Several technical factors affect agile development progress. These factors include source code versioning, unit testing (UT), acceptance testing (AT), continuous integration (CI), and releasing. These factors play a role in determining whether software produced for a user story (i.e. feature or use case) is 'working software' (i.e. the user story is complete) or not. One of the principles introduced by the Agile Manifesto [12] is that *working software is the primary measure of progress*. We propose that each of the technical factors impacts the progress towards working software; hence, they need to be managed. We will demonstrate that the outcome for each factor can be used to apply appropriate constraints and help determine the required co-ordination of the work of the software team to better manage the development progress.

In distributed agile teams, **informal methods**, such as video-conference meetings, can be used to raise the awareness of development progress [13]. However, with infrequent communications, it is difficult to understand how the work of one team member at one site influences the work progress of another team member at a different site. Team members may not recognise that there is an effect on progress or may not know who is affected. In addition, they may decide not to contact other team members, because of the time it takes to locate and notify the affected people.

Furthermore, **formal methods**, such as agile project management tools (e.g. Rally [14], Mingle [15], VersionOne [16], TargetProcess [17]), are widely used in distributed agile software development. These tools facilitate sharing progress information about iterations' tasks and user stories. The tools provide basic

progress status notifications. For instance, if a task is delayed, a team leader can be notified. However, these tools rely on team members' perceptions in understanding change in progress. Identifying and co-ordinating the impact of technical factors on development progress are not considered. For example, if modifying a source code artefact requires a further acceptance test to be developed, this will not be recognised by the project management tools.

The lack of mechanisms to effectively manage progress change resulting from the technical factors may lead to the project being delayed or to produce low quality code.

In our research investigation, I attempt to support the effective management of progress by providing a computer-based holistic approach to managing development progress that aims to explicitly identify and co-ordinate the effects of the various technical factors on progress. This will provide distributed agile teams with improved awareness of the actual progress of the project.

The holistic approach will help distributed agile teams determine change in progress as soon as it occurs. This can potentially reduce the testing bottlenecks at the end of each iteration and release, and can support better forecasting as it will be based on more realistic progress information.

The thesis argues that the computer-based holistic approach to managing progress is achievable and that it can overcome the limitations of the informal and formal methods.

1.2 Hypothesis, Aims and Objectives

The research presented in this thesis is based on the following hypothesis:

"*Managing development progress in distributed agile projects can be supported by providing a computer-based holistic approach that co-ordinates the impact of the different technical activities on development progress, and will provide improved awareness of the actual progress to team members*."

The hypothesis leads to the following aim and objectives for this research. The aim of the research is to develop a computer-based holistic approach to managing progress in distributed agile projects. The approach has to co-ordinate the impact of the various technical activities on development progress.

In order to achieve this aim, a set of research objectives are defined:

1. *Defining the concept of progress in the agile approach and the difference in progress tracking between the agile approach and the traditional (plan-driven) approach. This includes identifying the key technical factors affecting agile development progress.*

2. *Surveying how well the informal methods and the formal methods manage progress in a distributed agile development.*

3. *Identifying the co-ordination support required for managing development progress. This includes analysing the various events that cause change in progress.*

4. *Designing a computer-based system capable of providing the necessary co-ordination. Computer-based mechanisms have to take into consideration the impact of the technical activities on progress.*

5. *Evaluating the computer-based holistic approach. This includes preparing an evaluation methodology and determining whether the computer-based holistic approach is achievable.*

1.3 Achievements of the Research

The fulfilment of the objectives of this research will demonstrate the following achievements:

- Identification of the need for an effective approach to incorporate the impact of the technical factors (UT, AT, CI and Releasing, and source code versioning) on development progress. This is because these factors impact the progress towards working software.

- Definition of a computer-based holistic approach to manage development progress in distributed agile projects, to overcome the limitations of the informal and formal methods. The approach can identify the effects of change not only from the users (team members), but also from the various technical systems that cause changes in progress.

- Comprehensive analysis of how each of the technical activities may cause change in progress. Twenty-three progress change events are identified. For each of these events, an explicit identification of the co-ordination support required has been provided.

- A novel design approach is proposed for the design of a progress tracking system that takes into consideration the impact of technical activities on development progress. It enables the progress tracking system to keep track of the impact of the technical activities by placing them under control of the tracking system. This can be achieved by integrating the versioning functionalities into the progress tracking system and linking the UT tool, AT tool, and CI tool, with the progress tracking system. Four types of model are proposed to serve four different needs:
    - A novel version model is proposed that incorporates the outcomes of the technical activities to indicate the level of maturity of the source code versions associated with each task/user story.

  - A novel user story progress model is proposed. It provides better awareness of the progress state of user stories. The model reflects the impact of the technical activities on development progress. For instance, modifying a shared source code belonging to a completed story may lead to the story being incomplete. The modification effect on the story's progress has to be explicitly shown on the progress tracking system and reported to the affected team members.
  - A set of process models, covering all the technical activities is developed. Each process model clearly illustrates how a technical activity affects development progress. It also provides a suggested flow of activities for co-ordination support, including checking progress constraints, identifying the potential sources of progress change, finding and notifying affected team members when there is a progress change, and reflecting progress change in the tracking system.
  - A data model is proposed that represents the large number of dependencies among the different entities in the tracking system (i.e. tasks, stories, releases, unit tests, acceptance tests and integration tests). The dependencies can help identify how the development progress is affected by the technical activities and help target the co-ordination to those who are impacted by the technical activities.

- Development of a prototype system that is used to demonstrate that the computer-based holistic approach to managing development progress is sound and practical. The prototype system is used as a proof-of-concept for the holistic approach.

1.4 Structure of the Thesis

This section presents an overview of the organisation of the thesis. The first chapter has introduced the research undertaken, the hypothesis to be tested and highlighted the aims and objectives of the research and its original achievements.

Chapter 2: *Agile Software Development*

This chapter presents the main limitations of the plan-driven approach and provides a background to the agile approach. It also discusses how the concept of progress tracking is different in these two approaches. The key technical factors affecting agile development progress are also identified and discussed. This includes a survey that explores the popularity of the technical factors among the agile methods and includes a discussion of how they are used in agile development.

Chapter 3: *Managing Development Progress in Distributed Agile Projects*

This chapter first discusses distributed software environments. This includes the motivation for implementing distributed software development environments and the co-ordination challenge in such environments. The chapter also investigates the current approaches used to managing development progress in distributed agile environments. It discusses two main approaches: informal-based methods and formal-based methods. The analysis of the two approaches leads to suggesting a new approach (the computer-based holistic approach) to managing development progress in distributed agile projects.

Chapter 4: *Co-ordination Support Required for Managing Progress of Distributed Agile Projects*

This chapter identifies the co-ordination support requirements for managing progress of distributed agile projects. It analyses the progress change events that may result from performing the technical activities and provides explicit identification of the co-ordination support required to deal with these events. Two examples are provided to enhance understanding of the co-ordination support required.

Chapter 5: *Design of Progress Tracking System*

This chapter introduces a design approach for a computer-based progress tracking system. It describes an architecture that enables the tracking system to identify the impact of the technical activities on development progress. In addition, the four models mentioned in Section 1.3 (version model, user story progress model, process model and data model) are presented.

Chapter 6: *Evaluation*

This chapter evaluates the holistic approach to developing a progress tracking system proposed in this work. It discusses the evaluation methodology used and discusses three scenarios used for evaluation. It then describes developing a prototype system to validate the holistic approach.

Chapter 7: *Conclusions*

This chapter highlights the key aspects of the work, assesses the achievements against the aims and objectives and concludes with suggesting future work that could be carried out.

CHAPTER 2

Agile Software Development

This chapter presents the background to agile software development. It also discusses progress tracking in the agile approach and the various technical factors that affect progress of an agile development. The chapter starts by discussing the main limitations of the plan-driven approach in section 2.1 while section 2.2 gives background to the agile approach and the difference in progress tracking between the plan-driven approach and the agile methods. The key technical factors affecting agile development progress are then identified and discussed in section 2.3. The chapter concludes with a brief summary.

2.1 Limitations of the Plan-Driven Approach

Plan-driven methodologies (also known as heavyweight and traditional methodologies) have been widely adopted by software organisations for many years. The main common characteristics of these methodologies include providing thorough documentation, up-front system architecture and detailed plans [18]. The waterfall model [19], V-Model [20], rapid-prototyping model [21], spiral model [22], and the Rational Unified Process (RUP) model [23] are among the most popular plan-driven methodologies.

The waterfall process model has been widely used in both large and small software projects and has been reported as a successful approach especially for large and complex engineering projects in controlled environments [24] [25].

The waterfall model is a sequential phased-based approach to software development in which software is developed systematically from one phase to another in a downward fashion like a waterfall [19]. In the waterfall model, the desired functionalities of the software need to be specified beforehand. A detailed plan for the whole project is created at the beginning and the plan is then followed as precisely as possible. A common feature of the waterfall model is their emphasis on defining the scope, schedule, and cost of the project at the start.

The waterfall model has been severely criticised for its poor flexibility and lack of adaptability for requirements change. Somerville states:

"*Its major problem is the inflexible partitioning of the project into distinct stages. Commitments must be made at an early stage in the process, which makes it difficult to respond to changing customer requirements*" [25].

Customer requirements may change over time due to the rapid changes in the technology or the business environment [26]. In addition, the customer may not be sure exactly what requirements are needed before using a working prototype.

In projects using the waterfall model, the customer does not receive any software until the entire development is complete. If the software project runs over time or budget, it is likely that the final phase of software development will be left incomplete. Given the fact that the final stage is normally the testing and quality phase, this means that the most important development stage could be poorly carried out. Because defects and issues may remain for a long time before discovering them, they may rise over time and be harder to fix.

The V-Model has the same phases as the waterfall model but each phase is supplemented by verification and validation activities. The criticism of the V-

Model is that it is sequential and it divides development phases with sharp boundaries between them; this is the same problem as the waterfall model.

The rapid-prototyping model emphasises the building of an early prototype to help understand customer requirements. Similarly, the spiral model moves through a set of prototype builds to help the project team identify and reduce the major risks as early as possible. In these models, however, the prototypes may not be part of the design itself but merely representations that are thrown away after fulfilling their function; the majority of the design work carried out thereafter is performed in a similar manner to the waterfall model [27].

The iterative development approach builds the system incrementally; a few more features are added during each iteration until the entire system is completed. One of the most popular methodologies applying this approach is the Rational Unified Process (RUP). It is an iterations process framework that organises the development of the software into four phases (inception, elaboration, construction and transition), each consisting of one or more iterations. RUP requires producing large amounts of documents and although it is iterative, each iteration has to concentrate on the main emphasis of the phase it belongs to. That is, early iterations are mostly about defining requirements and architecture, while later iterations focus on implementation and testing [28].

The methodologies discussed above are heavyweight, document-centred and plan-driven approaches. Fowler [29] describes such approaches as engineering methodologies which may work perfectly for building a bridge but not for building software, as building software is unpredictable activity and hence could benefit from a different process.

2.2 Agile Approach

A more recent lightweight approach to developing software, called the agile approach, has emerged as a reaction to the limitations of the plan-driven approach. The agile approach proposes a different view of the certainty aspect of the software development process, compared to the plan-driven approach. In the plan-driven approach, intensive effort is spent in forecasting the customer requirements in order to reduce the number of changes. In the agile approach, the uncertainty in software development projects can be considered as a baseline assumption [30]. Thus, the agile software development approach can be regarded as a means of responding to uncertainty (adaptive), rather than as a means of achieving certainty (predictive) [30]. The agile approach focuses on 'reaction abilities', that is, the abilities to include changes late in the process rapidly and with low cost [31]. It does not try to avoid changes but it seeks to embrace them [32].

Agile concepts emerged in the mid 90s, when 'lightweight' software methods and techniques such as Extreme Programming (XP) (1999) [33], Scrum (1995) [34], Crystal Family of Methodologies (1998) [35], Dynamic Systems Development Method (DSDM) (1995) [36], Adaptive Software Development (ASD) (1999) [37], Pragmatic Programming (2000) [38], and Feature-Driven Development (FDD) (1999) [39] were independently developed.

The term 'agile' was agreed later, during a meeting when seventeen of the proponents of the "lightweight" methods came together in February 2001, in order to formalise common aspects of each others' methods. The outcome of the meeting was the production of the Agile Manifesto [12] which includes a set of values and principles forming the basis of the various agile methods.

2.2.1 Fundamentals of Agile Software Development

The Agile Manifesto identified four values for agile development. These are:

• *Individuals and interactions over processes and tools*

The agile development emphasises the relationship and the communality of the team members over using the heavy process models and tools.

• *Working software over comprehensive documentation*

Although the software documentation is useful in the development, the most effective documentation tool is the code itself. The agile approach stresses the point of keeping the code simple and straightforward so it can be easily understood.

• *Customer collaboration over contract negotiation*

Requirements are determined through customer co-operation and collaboration during iterative development, rather than setting these requirements in a strict contract at the beginning of the project.

• *Responding to change over following a plan*

In contrast to the plan-driven methodologies, the agile approach allows for the preparation of short term plans that are flexible to changes. The development group, comprising both software developers and customer representatives, should be well-informed, competent and authorised to consider possible adjustment needs emerging during the development process life cycle [40].

In the four points above, the manifesto recognises that while there is value in the items on the right, the items on the left are valued more. The participants pointed

out that "*the agile movement is not anti-methodology*" [12]. Highsmith, one of the contributors of the Agile Manifesto, states:

> *"We embrace documentation, but not hundreds of pages of never-maintained and rarely-used tomes. We plan, but recognize the limits of planning in a turbulent environment"* [41].

The agile values are described in more detail in twelve principles. These principles are listed in Appendix A. The principles are fundamental ideas that represent a high-level judgment on whether a software development method is agile or not. Abrahamson et al. [40] answered the question: *What makes a development method an agile one?* by providing the following characteristics of the agile approach:

- Incremental (small software releases, with iterative cycles),
- Cooperative (customer and developers working constantly together with close communication),
- Straightforward (the method itself is easy to learn and to modify, well documented), and
- Adaptive (able to make last moment changes).

2.2.2 The Concept of Progress in Agile Approach

Progress in the plan-driven methodologies is often based on the completion of deliverables such as the requirement specification document and analysis and design diagrams. It is difficult to judge progress based on these deliverables [25]. The progress reports may not reflect how healthy the project is. For instance, the progress report for a project that is at the end of the design phase may show that the project progresses well as all design diagrams are completed. However, team members may find many problems later in the integration phase or the testing phase. Software teams may struggle keeping all deliverables consistent when

change occurs. Additional time is spent in developing these extra artefacts that are not software.

The view in agile methods is different from the view in the plan-driven approach. Progress status is judged in agile methods mainly based on the essential output of the project which is basically the software that the customer will use. Principle 7 in the Agile Manifesto states that:

"*Working software is the primary measure of progress*."

Working software implies that the software is unit-tested, integrated and acceptance-tested by the customer.

The working software concept is easy to understand by the customer as well as developers. User stories represents the unit of progress measurement. If the customer is happy with the functionalities provided for a story (i.e. acceptance test passed), it is considered complete.

2.2.3 Extreme Programming

To better understand the agile approach, it is useful to describe one of the agile methods in detail and use it in the rest of this work as a representative of the agile approach methods. This will provide a common use of the terminology.

The method selected is Extreme Programming (XP) [42]. It has been widely acknowledged as the starting point of the various agile software development methods [40]. It includes the primary practices that have been adopted in projects using other agile methods. In addition, the literature shows that it is widely common for teams applying XP to use all the technical factors, mentioned earlier, in their projects (e.g. [174] [175]). This makes it a good choice to represent the agile methods in this research.

2.2.3.1 XP Process and Terminology

In XP development, requirements are described in terms of user stories, each of which represents a unit of functionality of the system (i.e. use case or feature). A release plan is created to determine how many user stories will be delivered to the customer in the next release. The user stories are distributed over several iterations; each iteration is completed in one week. Within an iteration, user stories are prioritised and broken down into tasks which are given initial estimates and then developed. After implementing a story, acceptance testing is done to ensure that what is implemented is what the customer wants. Figure 2-1 provides a general overview of the XP process.

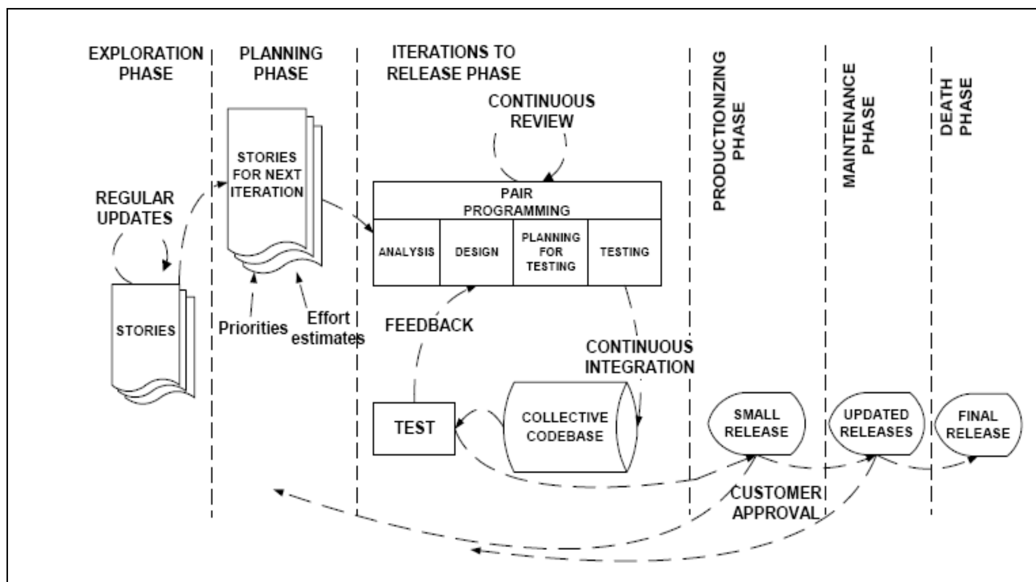


Figure 2-1. XP process model [40].

The fundamental XP terminology used in this work is described below.

User Story: A user story is the customer expression of a discrete feature of the system that will be discussed with other team members with the aim of transforming it into software. Stories are the primary input into the XP process.

Task: Stories are divided into discrete programming tasks assigned to developers during the planning session. Typically, each task should take a few days.

Unit Test: a test case or suite written to test the functionality embodied in a source code artefact (e.g. Java class).

Acceptance Tests: The customer writes acceptance tests for each user story. The acceptance tests describe what the user expects the system to do.

Release Plan: identifies what stories will be implemented over what period. The customer receives several releases during the project life. This is critical to getting valuable feedback in time to have an impact on the system's development. Each release can take several months before being submitted to the customer.

Iteration Plan: outlines what user stories will be implemented in one week. The customer chooses the most valuable user stories to be implemented in the iteration.

2.2.3.2 XP Values and Practices

Beck identifies five values for effective software development using the XP method. These are:

- *Communication*: XP emphasises the need for building a person-to-person, mutual understanding of the system under design through maximum face-to-face interaction.

- *Simplicity*: XP supports starting with the simplest design. Extra functionality can then be added later. It is believed that it is better to do a simple solution today and pay a little more tomorrow for change than to do a more complicated thing today that may never be used.

- *Feedback*: Developers obtain early feedback from the written code by writing unit tests and running integration tests. They also obtain feedback about the current status of the system when the customer performs acceptance testing.

• *Courage*: Beck states that XP teams must be courageous and willing to review the existing system and modify it, even if it is late in the project.

• *Respect*: Everyone on the team should feel appreciated or valuable. This will raise the motivation and will encourage loyalty toward the project.

As Beck says, "*Values bring purpose to practices*." and "*Practices are evidence of values*" [40]. XP values, described earlier, have been detailed in thirteen primary practices. The relevant practices to this research are discussed below.

Sit Together

The team is co-located in a single large room. This will encourage free conversations and simplify progress information exchange among team members.

This practice supports the sixth agile principle which provides that:

"the most efficient and effective method of conveying information to and within a development team is the face-to-face conversation".

Informative Workspace

The workspace has rich information about the project that can easily be observed by team members. The room has white shared boards and big charts showing information about project progress, such as status of user stories and acceptance tests (ATs).

Stories

It is the XP practice of thinking about software in terms of units of customer visible functionality. One or more sentences are written by the customer that captures what the customer would like to achieve. Each story is limited, and should fit on a small card to ensure that it does not grow too large. Stories are prioritised by the customer at the beginning of each iteration and then divided

into several tasks that are undertaken by different team members. Each story is normally accompanied with acceptance tests that determine when the story can be claimed to be complete.

Weekly Cycle

XP recommends reducing the short term planning cycle to one week. At the end of each weekly cycle (i.e. iteration), the XP teams normally complete an incremental version of the system. Tracking progress in the short-term allows better tracking of what has been completed.

Quarterly Cycle

XP also recommends having regular reviews of the high level system structure, its goals and priorities on a quarterly basis. This includes reflections on the team, the project and the progress (e.g. identifying project bottlenecks). At the end of each quarterly cycle, a new release of the working software is produced to the customer.

Continuous integration

Integration is one of the most difficult stages in traditional software development. This is because traditional development delays the integration process until the end of development. It will be easier if the software team adopts the practice of bi-weekly, weekly, or daily integrations.

In XP, after finishing every piece of work, it is recommended that it be integrated with the current system; hence, the system is built incrementally. Continuous integration allows for early detection of defects and conflicts, and contributes towards producing working software.

2.3 Technical Factors Affecting Agile Development Progress

This section identifies and discusses the various technical factors affecting agile development and their popularity in agile methods. It also highlights how agile teams use them in their projects.

2.3.1 The Technical Factors

Because agile progress is based on the 'working software' philosophy, it is wise to ask what factors contribute to producing working software. There are several technical factors that affect the progress of agile project development. These factors can be derived from the meaning of the term *working software*. Working software is recognised as the code that has been implemented, unit-tested, integrated and acceptance-tested [176]. Thus, activities involved in *unit testing*, *acceptance testing*, *continuous integration*, *source code versioning* may affect working software that are delivered to the customer during the *releasing* process. These factors apply to both traditional and agile projects; however, they represent crucial factors in agile projects due to the highly iterative nature of this approach and because agile development relies on 'working software' as a measure of progress.

Unit Testing (UT)

The developer has to produce well-tested code before the task is determined to be complete. Adding or modifying a unit test without running it or with a 'fail' result can affect the corresponding task's progress if it is already completed.

XP introduced the concept of test driven development (TDD). In this, developers working on tasks have to write the tests before coding the task. They have to produce well-tested code before a task is completed.

Several unit testing frameworks have been developed to automate and help simplify the process of unit testing, with support for a wide variety of languages. Examples of such frameworks include JUnit [44] for the Java language and NUnit [45] for the .Net language.

Acceptance Testing (AT)

Each story may have one or more acceptance tests and is not considered complete until all its acceptance tests pass. The corresponding completed stories may be affected if a new acceptance test is added to the story or an existing one is modified due to changes in customer requirements. The AT can be used as a measure of progress. Running Tested Features (RTF) [46] is a progress metric that uses the number of running ATs as an indicator of project progress.

Continuous Integration (CI)

CI is an effective way of identifying how healthy the overall code is at a specific point of time. The result of integration has a direct impact on development progress because it is a condition for completing stories.

There are two approaches used to provide the CI, synchronous or asynchronous:

- In the synchronous integration approach, every commit to the repository builds the system, as Martin Fowler suggests [47]. The main problem with this approach is that the build process may take a long time to succeed, which can delay sharing the code amongst the developers.
- In the asynchronous integration approach, developers share code that is either integrated or ready for integration. The integration might be done only once or twice a day. This provides a more flexible approach to the team members and is considered practical for broader situations such as distributed teams.

A further discussion of these two approaches and the strategies that each of them includes is provided in Chapter 5.

Releasing

Releasing is a special case of the integration process where a copy of the system is delivered to the customer. Releasing a user story requires that it is totally completed. This implies that the story's functionalities have been accepted by the customer.

Agile development emphasises the importance of releasing early and releasing often (RERO). That is, the customer is provided with multiple releases before producing the final product.

This allows a feedback loop between team members and the customer. The customer can say what they like and what they do not, and what stories they would like to see in the product.

The period between one release and another is normally different between agile projects and is based on several factors, including the size of the project and customer preferences. However, the Agile Manifesto recommends providing releases as soon as possible. The third principle states:

> "*Deliver working software frequently, from a couple of weeks to a couple of months, with a preference for the shorter timescale*."

The release process takes place at the end of an iteration. '*Potentially shippable code*' is produced and hence a potentially releasable version of the system becomes available to the customer.

It is worth mentioning that the output of a release process is not necessarily a *released* version but could be only a *releasable* version. A version of the system can be placed in a test environment. This test environment is as similar as

possible to the real production environment. The decision to put the releasable version into production is for the customer.

Here, we will focus on the production of releasable versions of the system, regardless of whether a release is deployed into the business environment or not.

Source Code Versioning

Creating, modifying or deleting some source code artefacts will usually change the actual project progress. There are many cases where changing the source code influences project tasks, user stories and releases. For instance, modifying a source code version that belongs to a completed story means that the story is no longer deemed to be complete.

Agility is about creating and responding to change [32]. For this reason, most agile methods recommend software configuration management (SCM) tools to automate the change process. According to Cockburn [48], in Crystal methods, versioning and configuration management tools are "*the most important tools the team can own*." Agile methods consider the ability to revert to earlier versions of development artefacts highly valuable [49]. Since rapid development and quick changes may lead to mistakes in development, it is important that earlier versions of artefacts are accessible [49].

Ron Jeffries et al. [50] stress that, for agile teams, there should be as few restrictions as possible in an SCM tool; for example, there should be no password, no group restrictions, and as little "hassle" as possible. This is supported by the experiences of Lippert et al. [51], who found that optimistic concurrency control is a superior locking mechanism in agile methods.

Key technical activities affecting development progress are shown in Table 2-1.

Unit Testing (UT)	Acceptance Testing (AT)	Continuous Integration (CI) & Releasing	Source Code Versioning
• Create a new UT • Update existing UT • Delete UT • Run UT	• Create a new AT • Update existing AT • Delete AT • Run AT	• Perform integration • Make a release	• Create an artefact • Modify an artefact • Delete an artefact

Table 2-1. Key technical activities affecting agile development progress.

Task progress is usually linked to whether functionalities involved in it are unit-tested or not, whereas story progress status also covers integration level and acceptance test level (see Figure 2-2).

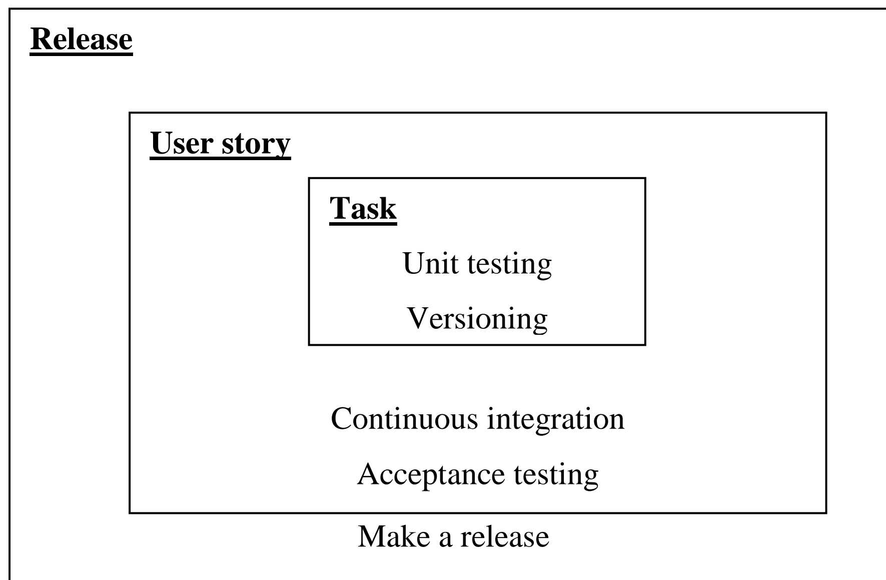


Figure 2-2. Level of influence of the technical factors.

2.3.2 The Technical Factors in Agile Methods

The popularity of the technical factors among the agile methods was surveyed. This has been done by investigating the essential reference for each agile method where the formal description of the method is introduced (XP [42], Scrum [57], Crystal Family of Methodologies [78], DSDM [36], ASD [37] and Pragmatic Programming [38]). The results showed that most agile methods recommended using UT, AT, CI, releasing, and source code versioning (see Table 2-2). Most methods have explicitly mentioned them in the formal methods description.

Agile Method	UT	AT	CI	Releasing	Versioning
Extreme Programming	●	●	●	●	○
Scrum	○	○	○	●	○
Crystal Family of Methodologies	●	●	●	●	●
Dynamic Systems Development	●	●	●	●	●
Adaptive Software Development	○	○	○	●	●
Agile Modeling	●	●	●	●	●
Pragmatic Programming	●	●	●	●	●
Feature-Driven Development	●	●	●	●	●

● Explicitly mentioned ○ Implicitly mentioned [1]

Table 2-2. The technical factors in agile methods.

However, not all the technical factors have been mentioned explicitly in some agile methods:

- Source code versioning is implicitly mentioned in XP
- UT, AT, CI and source code versioning are implicitly mentioned in Scrum

[1] A technical factor may not be explicitly mentioned by an agile method but some practices used by the method requires doing the technical factor (e.g. the *Ten-Minute Build* practice in XP implies that there is a version control system where the most recent source code versions can be retrieved from).

- UT, AT and CI are implicitly mentioned in Adaptive Software Development

XP did not explicitly emphasise the importance of using systems to manage source code versions in XP projects. Paulk [52] states that SCM is partially addressed in XP via collective ownership, continuous integration and small releases. However, the literature on XP emphasises clearly the need for versioning systems (e.g. [46] [53]).

Furthermore, the focus in Scrum and ASD is not on the development techniques. Scrum has focused on providing a project management framework, while ASD's primary focus is on the problems of developing large and complex systems. Scrum and ASD provide very few practices for day-to-day software development work [40]. These methods state that they welcome practices from other methodologies for use in the development.

Regardless of the agile method applied, the literature and survey show that UT, AT, CI, releasing, and source code versioning have been widely adopted in agile projects (e.g. [54] [55]).

2.4 Summary

This chapter has addressed three key points. First, the limitations of the plan-driven approach were described. The main limitation is that both the technology and the business environment keep shifting during the project life, and, hence, the requirements may get out of date within even a short period of time.

Secondly, the agile approach, with XP as an example, has been presented. How the agile approach overcomes the limitations of the plan-driven approach was demonstrated. The concept of progress in the agile approach was introduced.

Contrary to the plan-driven approach that is based on producing deliverables to measure progress, the working software is the primary measure of progress. Working software is the code that has been implemented, unit-tested, integrated and acceptance-tested.

Finally, the key technical factors affecting agile development progress have been identified and discussed. These are unit testing, acceptance testing, continuous integration, releasing and source code versioning. The influence level of these factors has been analysed. In addition, the extent to which these factors have been mentioned in the various agile methods has been explored and discussed. The result shows that most agile methods explicitly mention the technical factors.

CHAPTER 3

Managing Development Progress in Distributed Agile Projects

Distributing agile software development over multiple sites has gained a noticeable interest in both the literature and in industry. A common problem in projects not co-located is managing development progress. This chapter looks at the current approaches used to manage development progress in distributed agile environments. It discusses two approaches: informal methods and formal methods. The informal methods rely mainly on humans to manage progress while formal methods utilise automatic mechanisms in storing, retrieving and manipulating progress information to achieve the goal of managing progress. The analysis of these approaches shows that they are insufficient and, as a result, a new approach is suggested.

3.1 Managing Progress of Agile Software Development

Progress management is commonly understood as a managerial task that is used to provide information about the project's progress. Project management implies tracking and monitoring processes to observe project tasks, so that potential problems can be identified in a timely manner and corrective action can be taken, when necessary, to control the execution of the project [56].

As progress status is judged in agile methods based on the working software, it is required to monitor the status of the software and manage the technical activities that affect it. The source code versioning, unit testing, acceptance testing, continuous integration and releasing imply technical activities have an effect on the working software progress. Being aware of the actual progress of the agile project is not only important for the project manager, but also for the team members. This is because the progress is a result of highly interdependent tasks. Any task carried out by a team member may have an effect on the progress of other tasks carried out by other team members.

XP supports managing progress of co-located teams by two main practices: *Sit Together* and *Informative Workspace*. When team members sit together, they are expected to share information about factors that may affect the progress of the project through face-to-face communication. The ad-hoc co-ordination is likely to facilitate partial sharing of the progress information amongst team members.

XP teams are also encouraged to surround the workspace with rich information about the state of tasks, stories and tests that are updated continuously. They often use big charts to visualise the development progress. A commonly used chart is the burn-down chart [57] (Figure 3-1). The solid line on the chart shows the actual remaining work, while the dashed one represents the planned remaining work.

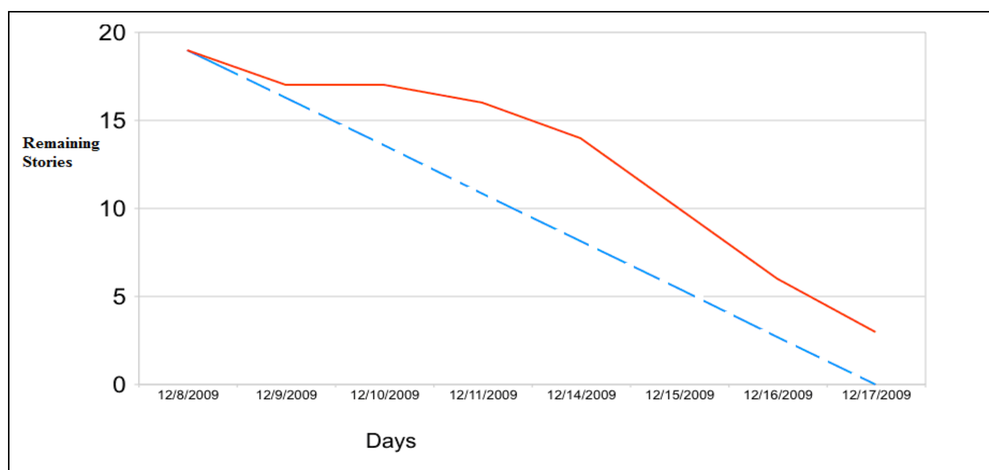


Figure 3-1. Burn-down chart.

When the project is distributed, team members find it harder to maintain an awareness of how the technical factors are affecting the progress of their tasks. Before discussing how the distributed agile teams manage development progress, a brief background about distributed software development is provided in the next section.

3.2 Distributed Software Development

Distributed software development (also known as Global Software Development and Multi-Site Development) means that the software project does not take place in one site but in several places, where stakeholders involved in the process are physically distant [4]. The practice of distributing software development has rapidly increased during the last two decades [1] [2]. This section will discuss the main motivations for implementing distributed software development and will also discuss the need for co-ordination support in such environments.

3.2.1 Motivations for Distributing Software Development

There are several reasons for the shift toward developing software remotely. The often cited drivers are those identified by Carmel [3]:

- *Reducing costs*

  Software companies can reduce the cost of developing software by performing the development in countries where the workforce is cheaper. In addition, countries differ in business-tax rates. While they are high in western countries such as the UK, other countries provide tax benefits to companies which start development centres in their country or even provide funding to increase local business [58].

- *Customer Distribution*

  A customer's business might be distributed over several branches. It can be beneficial to be close to the customer, or at least to build localisation points in close proximity to the markets in order to obtain information about the local markets [59]. If team members need to be close to the customer in some or all the branches, a multi-site software development project is required.

- *Promise of round-the-clock development*

  Software companies can benefit from doing the project globally to reduce the overall time of the project. Assuming that there are two teams, one in the United States and the other in India, the company can obtain 16 working hours daily. Completing a product and delivering it to the customer in a short time can be a distinctive advantage. Therefore, companies strive to reduce the time-to-market (TTM) value of their product to the lowest possible.

- *Limited pool of trained workforce*

  Some software projects might prefer to have the expertise ready rather than having to lose time in training team members for a particular project. If the expertise is not available locally, this could force the company to look for it in distant places, making the development project distributed.

Furthermore, there are circumstances that make creating co-located teams difficult. Examples of such circumstance include [60] [61]:

- Office arrangements may not allow the whole team to be situated at one location.

- The team might be too large to fit into one location and even if this was possible, then communication would produce a high level of noise.

- New models of work such as Tele-work explicitly demand distribution.

- Distribution can minimise the risk in case of natural catastrophes or other unexpected events.

3.2.2 The Need for Co-ordination Support

The perceived benefits of distributing software development are diminished by several challenges, which have been intensively discussed in the literature. In spite of the development methods applied, co-ordination is one of the primary problems of developing software on multiple sites [3]. Herbsleb and Grinter observe that co-ordination problems were greatly enhanced across sites, largely because of the breakdown of informal communication channels [62]. The temporal, geographical and socio-cultural barriers impose a co-ordination challenge to distributed teams [4] [3].

Frequency of communication generally drops off sharply with physical separation [10] [11]. Inadequate communication among team members causes reduced response times and irregular information flow. Consequently, co-ordination problems result in frequent delays and re-work. Time-zone differences further worsen the situation as it reduces the time-window for effective synchronous communication between remote teams [65].

Co-ordination problems could be exacerbated if the distributed teams share different cultures. Language difference, attitudes, and communication styles may negatively affect distributed teams. Studies show that distributed teams that share different cultures may not be as cohesive as local ones [66].

Furthermore, the co-ordination problem shows a positive relationship with the degree of interdependencies between the distributed sites. Little co-ordination

difficulty is expected in projects that use offshore outsourcing, where little work is shared between the onshore and offshore teams. In contrast, projects that are formed from fully dispersed members are likely to encounter high co-ordination overhead.

Distributed software projects using agile methods are likely to encounter more complex co-ordination problems, because agile methods are aimed at co-located projects, where teams rely on intensive communications among team members to co-ordinate their work. However, there are an increasing number of agile organisations working remotely to gain the advantages of distributing the work [5]. With the absence of face-to-face interactions, numerous co-ordination difficulties are reported (e.g. [6–8]). One of these difficulties is how to manage development progress of distributed agile projects. The next section discusses how agile projects currently deal with this issue.

3.3 Current Approaches to Managing Progress in Distributed Agile Projects

The primary methods used by distributed agile projects to manage development progress can be divided into two approaches: informal methods and formal methods. These approaches have been extensively reviewed for this section. A roadmap of the various methods discussed is given in Figure 3-2.

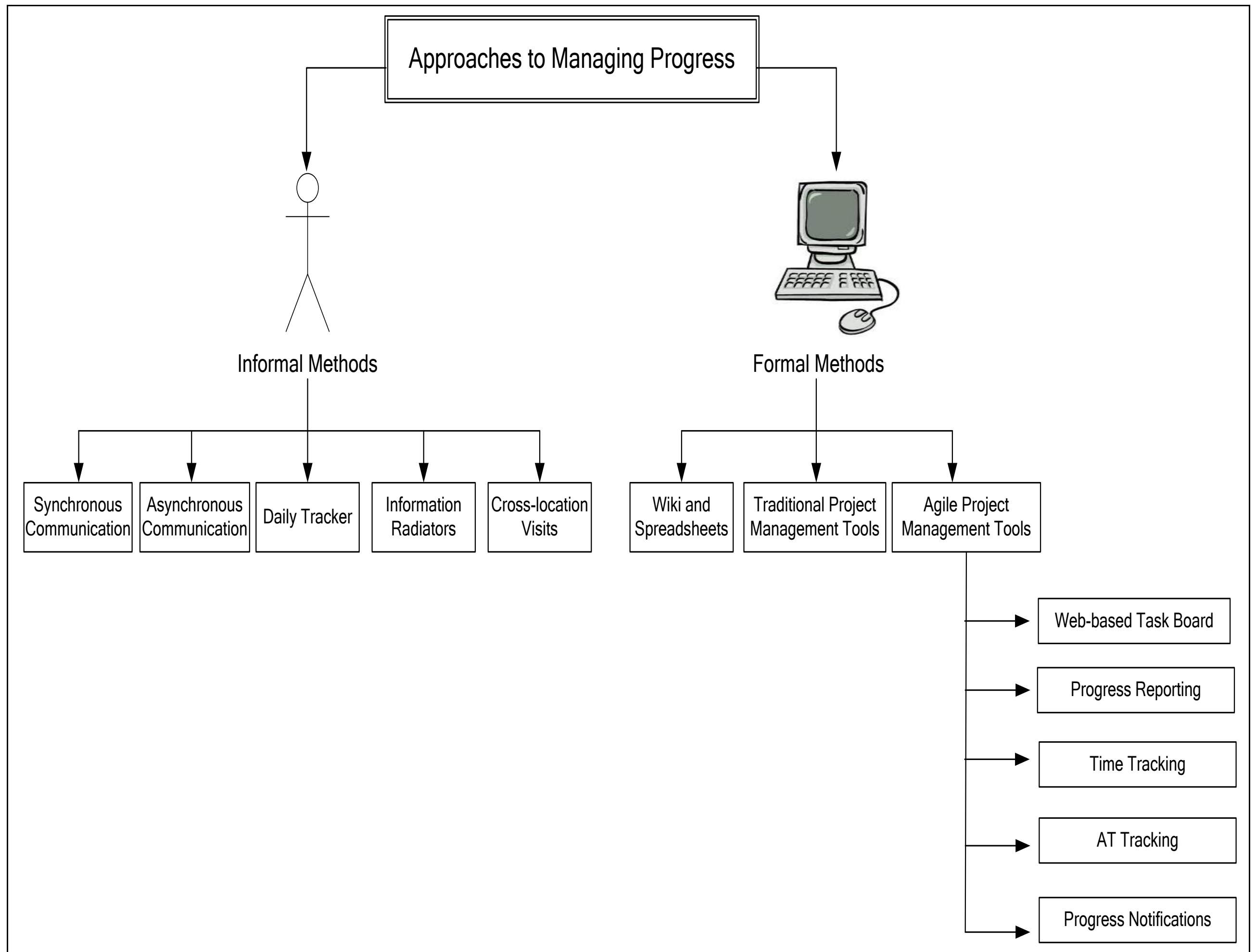


Figure 3-2. A roadmap for the current approaches used for managing progress

3.3.1 Informal Methods

Distributed agile teams use several informal methods to track progress information. The main informal methods are synchronous communication, asynchronous communication, daily tracker, information radiators and cross-location visits. These are discussed below.

3.3.1.1 Synchronous Communication

In co-located teams, face-to-face communication and daily stand-up meetings enable team members to easily share progress information. In distributed teams, meetings can be held by synchronous tools such as audio and video-conferencing tools. Stand-up meetings may be held for about half an hour everyday using these tools. Other meetings can be scheduled weekly and monthly.

In addition, some teams may use instant messaging (IMs) for one-to-one communication between team members. These are likely to be used in situations where developers need to communicate personally about issues such as coding aspects or design aspects or any clarifications.

A large number of case studies about distributed agile projects reported difficulties in using synchronous communication (e.g. [67] [68] [69]). Some of these difficulties are:

- Good video-conferencing tools can be expensive for teams with a limited budget.
- Meetings have to be planned in advance to ensure that all involved team members are available and can participate.
- Cultural and language differences may reduce the participation among team members (i.e. some team members may keep silent).
- Team members may get exhausted with long teleconferences.
- Some teams report spending significant time in resolving technical issues, such as sound quality not always being good enough due to limited bandwidth.
- Teams may encounter difficulties in recognising speakers when not seeing their faces or when a large group of team members participates in a meeting using a single camera.
- Different cultures often have different public holidays at different dates. Moreover, people of different cultures prefer to take holidays at different times of the year.

- Time-zone differences cause challenges to arrange meetings. Having multiple time zones may only provide little time overlap in which team members can interact simultaneously.

These issues may cause synchronous meetings to be held less frequently than physical stand-up meetings in co-located projects. Thus, distributed teams may prefer using synchronous tools only for the major progress update events.

3.3.1.2 Asynchronous Communication

Due to the many issues associated with the synchronous communication approach, distributed teams may rely more on asynchronous communication tools, such as e-mail and community discussion boards. While e-mail is more direct and chiefly used for point-to-point communications, community discussion forums are more open and allow interested people to subscribe to the list [70] [71].

The asynchronous tools are cheap, popular, and have fewer technical issues. A further advantage of these tools is that they allow information to be shared without having to schedule meetings [72].

Layman et al. [73] had positive experiences using e-mail to share information. Their findings indicate the importance of short, asynchronous communication loops that can serve as a sufficient substitute for synchronous communication. They recommended providing timely response to developer inquiries to prevent affecting development progress while awaiting a definitive answer. On the other hand, empirical evidence indicates that increasing reliance on asynchronous communication channels can result in higher software defect rates [74].

Kajko-Mattsson et al. [8] observed that the use of tools such as email proved to be insufficient for maintaining the daily communication as dictated by the agile values. Using these tools results in a slow turnaround in communication [75] and often causes misunderstandings, due to messages being composed quickly [76].

In addition, managing e-mails and filtering them may become more difficult and burdensome over time as the team and project knowledge grow in size and complexity [77]. It is also expected that some of the shared information will be misunderstood because of culture and language differences and because the body language, voice inflection and emotions are lost through this type of communication.

3.3.1.3 Daily Tracker

The daily tracker's role has been used in many distributed agile projects. A couple of times a week, the tracker finds out where everyone is with the iteration [43]. He tracks the individual progress of the developers by asking them how many days they have worked on the tasks and how many more days are left to complete them.

The daily tracker's role helps reporting progress, but does not support the management of the daily dependencies among team members' work, which may affect development progress.

3.3.1.4 Information Radiators

Cockburn suggests having an 'information radiator' in the workspace [78]. An Information Radiator is a screen displaying information (e.g. progress information) in a place where passers-by can see it. It shows team members information they care about without having to ask anyone questions. Examples of the displayed information include burn charts, and state of acceptance tests. Two

characteristics are key to a good information radiator: the information must change over time and it takes very little energy to view the display [78].

Similar to the daily tracker practice, the information radiator can support sharing the daily progress information but it cannot support identifying and managing changes in development progress.

3.3.1.5 Cross-location Visits

Cross-location visits have been frequently recommended for distributed agile projects (e.g. [78] [79]). Team members are often rotated across project locations, to work within multiple teams. This helps in solving conflicts and misunderstandings among the distributed teams.

In addition to the cost constraint of this practice, the visits do not serve the aim of sharing and managing the daily progress information but only help sharing of the overall progress information.

3.3.2 Formal Methods

Distributed agile teams use several formal methods to manage development progress. The key formal methods include Wikis and spreadsheets, traditional project management tools, and agile project management (APM) tools. This section will discuss these methods.

3.3.2.1 Wikis and Spreadsheets

The basic technologies used for managing development progress are Wikis and spreadsheets. These tools allow users to freely create and edit content.

The advantage of using Wiki-based systems is that they provide a visible environment, making it easy to check project status, update task lists and view the team members' work progress [80].

Furthermore, online spreadsheets such as Google Spreadsheets [81] allow distributed team members to share and edit the same file at the same time, providing different editing permissions. They can also produce burn-down charts automatically.

However, Wikis and spreadsheets have limitations. Dubakov and Stevens [82] state that "the problem with Wiki and Excel is quite common … they do not have business logic behind them, but provide frameworks to resolve simple data manipulation problems." They observe that these tools provide little support for working on distributed environments and limited support for progress reporting and progress visibility [82].

A case study applied to a geographically distributed team using a wiki-based system called MASE [83] revealed further problems. When many minds collaborate together in a Wiki repository, it becomes more difficult to search and maintain as users contribute more and more content into the repository over time. In addition, content albeit useful may be put in the wrong place.

3.3.2.2 Traditional Project Management Tools

Traditional project management tools such as MS Project [84] could be used with agile methods. These tools can show information in PERT charts, Gantt charts and work breakdown structure charts.

Based on the surveys in [54] and [85], traditional project management tools have been utilised to manage development progress by many distributed agile projects. Most project managers are familiar with traditional tools and it is easier for them

to manage iterations using well known tools [82]. Main advantages include ease of use, flexibility and workflow support [86].

Unlike traditional software projects, only a part of the requirement is known when the project starts and new requirements will constantly emerge during development; this makes it unfeasible to follow the progress of the development work with these traditional tools [87]. Recreating the traditional charts whenever a new requirement emerges would take resources out of development work [87].

These tools are not designed for agile development and hence they do not include key progress tracking features, such as burn-down charts and story/task boards.

3.3.2.3 Agile Project Management (APM) Tools

Due to the limitations of the previous tools, a new generation of project management tools are being developed to satisfy the agile approach (e.g. Rally [14], Mingle [15], VersionOne [16], TargetProcess [17]). A review of thirty APM tools (Table 3-1) revealed a number of different mechanisms available to assist in supporting the management of distributed agile development progress. The review includes both commercial and open source tools and covers the most popular APM tools according to the surveys in [54] and [85].

In order to provide a comprehensive review of the available mechanisms and how they are used, the review has been carried out using a number of methods:

- working on trial versions offered by the surveyed tools.
- watching demos explaining the tools' functionalities.
- reading the formal description of the tools. This is normally made available as a text in the software website or as white papers written by the software company.
- asking direct questions through community boards associated with the software websites.

APM tools are be used over the Internet either directly by the browser or by web-based applications. Through these tools, it becomes easier to share progress information among the distributed agile teams. The key progress tracking mechanisms in these tools are web-based task board, progress reporting, time tracking, acceptance testing (AT) tracking and progress notifications. These are discussed below.

Agile Tool	**Web-based Task-board**	**Time Tracking**	**AT Progress Tracking**	**Notifications about Task**	**Notifications about Story**	**Notifications about AT**	**Progress Reporting**			
							Iteration Burn-down	**Release Burn-down**	**Velocity**	**CFD**
Rally [14]		●	●	●	●	●	●	●	●	●
Mingle [15]	●	●	●	●	●	●	●	●	●	●
VersionOne [16]	●	●	●	○	○	○	●	●	●	●
ScrumWorks [88]	●	●		●	●		●	●	●	
ExtremePlanner [89]	●	●	●	○	○		●	●	●	
XPlanner [90]		●		●			●		●	
TargetProcess [17]	●	●	●	○	○	○	●	●	●	●
Pivotal Tracker [91]	●	●			○			●	●	
Scrum VSTS [92]	●		●				●	●	●	●
Agilefant [93]		●					●		●	
IceScrum [94]	●		●		●		●	●	●	●
Planbox [95]		●			○		●		●	
XP StoryStudio [96]							●		●	
XPWeb [97]									●	
AgileWrap [98]	●	●		●	●		●	●	●	●
ScrumDesk [99]	●	●					●	●	●	
SpiraTeam [100]		●	●	●	●	●	●	●	●	
Leankit [101]	●			○	○					●
DevSuite [102]	●	●	●	●	●	●	●		●	
TinyPM [103]	●	●		●	●		●	●	●	●
Planigle [104]	●				●		●	●	●	●
Acunote [105]		●		●			●			
On Time [106]	●	●		●	●		●			
AgileZen [107]	●									●
ScrumPad [108]	●	●	●	○	○	○	●	●	●	
eXPlainPMT [109]			●						●	
AgileBuddy [110]		●					●	●	●	
Daily Scrum [111]	●	●					●	●	●	
Express [112]	●	●					●		●	
Agile Tracking [113]							●		●	

Key: ● Full support for a mechanism. ○ Partial support for a mechanism.

Table 3-1. A review of progress tracking mechanisms in APM tools.

3.3.2.3.1 Web-Based Task-Board

Task-boards are commonly used in co-located teams to visibly show the progress of tasks/user stories. They show all user stories with their tasks for the current iteration. Usually, each user story and task is represented by cards stuck to a board. Distributed teams use a web-based version in imitation of the manual task-boards with easy drag-and-drop facilities (Figure 3-3).

The task board usually has three main columns:

- Un-started (To Do): this holds all tasks that are not done.
- In Progress (In Process): a task is moved to 'In-Progress' state when a developer starts working on it.
- Done: a task is moved to 'Done' state if the functionalities required for the task have been accomplished.

Story	To Do	In Process	Done
1 - Story 1 3 point(s)	1-2 write business logic 20 hour(s) remaining 1-3 test! 9 hour(s) remaining	1-1 design UI 10 hour(s) remaining	
2 - Story 2 7 point(s)			

Figure 3-3. A typical web-based task-board

Nineteen of the tools reviewed have a graphical representation of task-boards while the rest allow merely for textual representation of a task's status. Tools such as On Time [106] and VersionOne [16] enable users to add extra columns such as 'To Be Verified' and 'Tested', which can show more detailed progress information.

3.3.2.3.2 Progress Reporting

During each iteration, while the team members are focused on creating the new user stories they have committed to deliver, the project manager is responsible for understanding the progress that the team is making and keeping the customer informed of any potential delays in the development. Most APM tools provide users with graphical reports that show key progress information about the project. These reports include:

- Iteration Burn-down Chart (the work that needs to be completed over an iteration).
- Release Burn-down Chart (the work that needs to be completed over a release).
- Velocity (number of units of work, i.e. user story points, completed over a period of time).

While velocity concerns the work done and how fast it is being done, the burn-down charts allow for forecasting. They allow "what if" analysis to be performed by adding and removing functionality from the release to get a more acceptable date or extend the date to include more functionality [57].

The review revealed that, of the 30, 25 APM tools provide iteration burn-down charts, 17 tools provide release burn-down charts and 24 tools provide automatic calculations of the project's velocity. Some tools, such as eXplainPMT [109], has burn-down charts but it is based on the whole project, not for each iteration nor each release.

Further progress charts called Cumulative Flow Diagrams (CFDs) [114] are offered by 11 APM tools. CFDs are constructed by counting the number of user stories that have reached a certain state of development at a given time. CFDs provide further detailed information about the 'Work In Progress' (WIP) state. Common progress points measured are: designing, coding and testing.

3.3.2.3.3 Time Tracking

Another progress tracking feature offered by the majority of APM tools (21 tools) is time tracking, in which hours spent/hours remaining for each task/story/iteration are presented. It replaces the tracker role in the co-located teams mentioned in the informal methods. Instead of having every team member entering their time, the time is calculated simply based on when a team member changes a task's status to 'In Progress', and when he sets the task to 'Done'.

The online derivation of time tracking data supports distributed agile projects as team members are scattered over different sites. It also eliminates erroneous data and time wastage problems existing in the manual calculation method.

3.3.2.3.4 AT Progress Tracking

Eleven of the APM tools reviewed allow scheduling and tracking acceptance tests' progress during the different iterations. Feedback on tests' progress is provided by a built-in electronic testing board and through various types of AT graphical reports. Examples of these reports include the test run progress rate graph produced by the Scrum VSTS tool [92] (Figure 3-4).

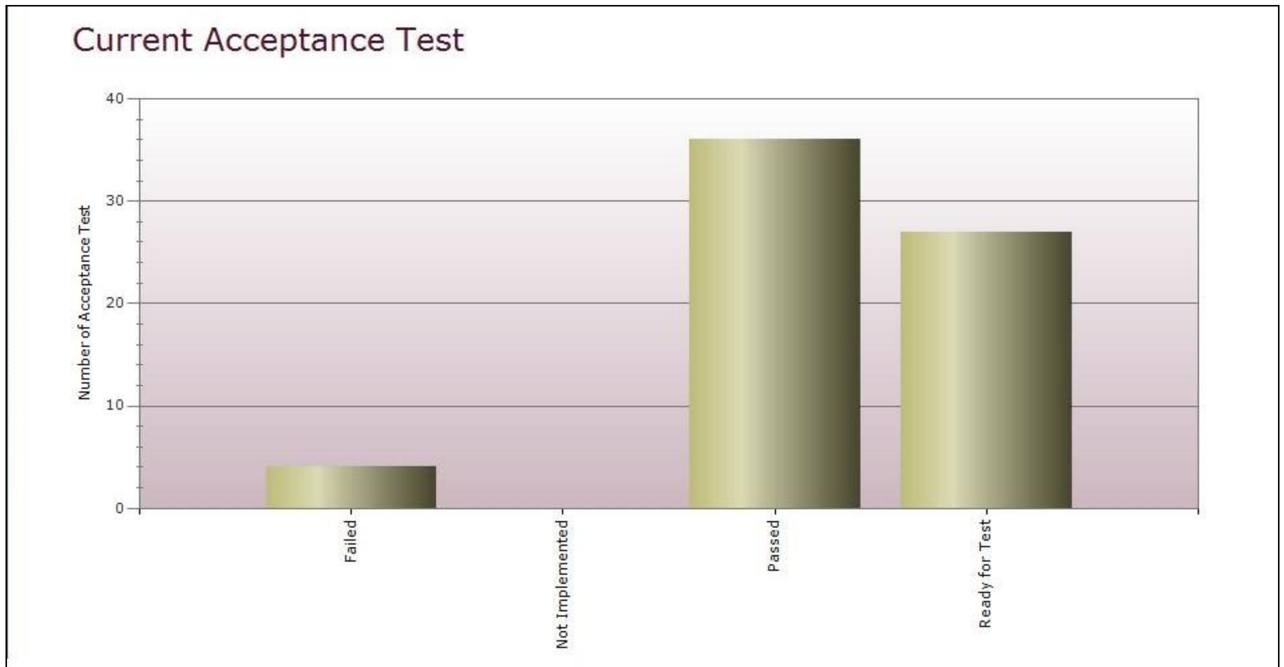


Figure 3-4. Test run progress rate graph by Scrum VSTS [92].

Acceptance tests are linked with their corresponding user stories. In addition, some APM tools such as VersionOne maintain a full history of each acceptance test which can be used for traceability purposes.

3.3.2.3.5 Progress Notifications

An effective progress notification system is an important requirement for managing development progress in distributed agile teams. Team members have to be made aware of the changes in development progress that affect them. Many of the APM tools reviewed provide some support for progress notifications when there is a change in progress status. Fourteen of them provide notifications when the progress status field of a task is changed by a team member, while 16 tools notify when the progress status field of a user story is changed. In addition, 6 tools only provide notifications if there are changes in the progress status field of an acceptance test.

Rally [14] allows team members to set up personalised notifications. They can select the types of event to be notified. Notifications offered by Mingle [15] are classified into three main types: user-generated messages between team members, system-generated alerts from subscriptions, and admin-level announcements to the whole project team. For instance, team members can send messages to raise awareness about new issues or provide immediate visibility of the status change of an asset (i.e. task, story or test). Team members can also set up subscriptions that will alert them if there are changes to a specific asset.

ScrumWorks [88] allows team members to select the period at which they wish to receive notification of changes. Notifications can be sent either immediately when each change occurs, or a once-daily listing of all accumulated changes.

Less robust notification systems are provided by VersionOne and TargetProcess. In VersionOne, team members cannot subscribe to events. Only the asset owner is notified when change occurs. The notification system in TargetProcess is role-based, that is, selecting any of the system roles will send the notification to all members of that particular role in the appropriate project. For example, selecting developer will notify all team members whose project role is developer.

The notification system in Planbox [95] is also limited. The scope of notifications is restricted to user stories only (called *items* in Planbox). Moreover, Planbox does not offer an event subscription service. A team member is notified either when the progress status field of the stories he works on has been changed, or when the progress status field of any story in the project has been changed. Notifications in Planbox can be triggered by various activities including progress status changes. Conditions can be defined so that only business critical items result in an email notification being sent, for example when an item’s status is changed to ‘Done’.

Agilewrap [98] sends notifications if a task or user story is overdue. In addition, if somebody accepts (story passed testing) or rejects (story did not pass testing) a user story, the story owner is notified.

ScrumDesk [99] does not provide notifications about specific assets. However, when the system starts, it displays all changes since the last time the user logged out.

In ScrumPad [108], the asset creator can designate the team members who will receive notifications about change in the asset's progress. This could be a disadvantage as the creator may not know who is affected by his work.

3.4 Justification for Computer-Based Holistic Approach

In the informal approach, managing progress of distributed agile teams is conducted in an *ad hoc* manner by the individual team members. If a change in progress has been introduced, the originator of the change has to co-ordinate the introduction of the change with other team members affected. A significant limitation of the informal methods is that the impact of the change may not be fully recognised by the team members. This is because of the difficulty of understanding how the work of one team member at one site influences the work of another team member at a different site. Team members may not recognise that there is an effect on progress or may not know who is affected. In addition, they may decide not to contact other team members, because of the time it takes to locate and notify the affected people.

The formal approach uses several mechanisms, incorporated into computer systems such as APM tools, that can be used to facilitate managing development progress. The distributed team members use these mechanisms to register, share and report the progress information. However, the main limitation of the formal approach is that the computer systems are static and rely completely on team members to report changes in progress. A team member performs a task and then registers the task's progress status in the computer system. Changes in progress caused by technical factors mentioned in the previous chapter, e.g. modifying

source code, are not logged by the formal approach; hence, if these changes affect development progress, this will not be discovered.

From the analysis of the informal methods and the formal methods, it is clear that these approaches are insufficient to fully identify and co-ordinate changes in progress caused by the technical activities. Although most distributed agile projects combine methods from both approaches to manage development progress, the literature shows that the distributed teams still have difficulties. Sauer [9] points out that progress status is less visible and controllable in distributed agile projects. Peng [10] observes that "teams have a difficult time keeping track of progress" in a distributed agile project. Teams may end an iteration having a large number of failed acceptance tests, delivering progress information late and to the wrong team members. Jeff Patton, a team leader in several agile projects, states that he noticed many agile organisations struggling to keep track of the acceptance tests. He states [11]:

"*When an acceptance test fails, it's usually a long time after the offending code has been checked in. In fact, a lot of code may have been checked in. This makes finding the offending code difficult. Also, it's not always clear who should be finding and fixing the issue. It's not the person who wrote the test, if his is a role that writes tests and not code. It's not clear which developer should fix the code*."

Better progress management support can be achieved by providing a computer-based holistic approach to developing a progress tracking system. The progress tracking system has to have a holistic view from the perspective that it needs to realise the effects of changes not only from the user (team members), but also from the various technical systems that cause changes in progress.

This will first require analysis of the various events that cause change in progress. This includes identifying the co-ordination support necessary for managing these events.

The holistic approach will also require designing computer-based mechanisms that take into consideration the impact of technical activities on progress. This means that there must be connections between the tracking system and the technical systems.

The proposed computer-based holistic approach responses to the distributed agile development literature which highlighted the need for providing more formal mechanisms (i.e. such as formal systems to track progress) to co-ordinate distributed teams (e.g. [5] [172]). A main reason for recommending these mechanisms is to reduce the need for informal communication due to its limitations in distributed environments.

Recently, several APM tools have started providing integration with some technical systems (UT tools, AT tools, versioning systems, CI tools). For instance, Rally, TargetProcess and VersionOne, provide integration with several commercial versioning systems. These integrations allow developers to synchronise their updates to tasks and source code without taking additional time to log their activity into both of the systems. They are also integrated with the UT tools to provide test tracking. However, these integrations are fairly simple and solely provide a linkage between the tracking system and the technical systems. This is insufficient to manage the impact of changes from technical activities on development progress.

Asklund et al. [115] mention the need to integrate source code changes to progress tracking data. They suggest adding task and story numbers as a comment with every check-in. Appleton et al. [116] support this by pointing out that "one of the most basic ways to help connect and navigate information is with a task-based approach [task-level commit] that links every action and event in the version-control system with a corresponding action and event in the tracking system." However, these methods do not provide automatic identification of potential changes that affect development progress and do not support managing the impact of changes.

Traceability tools have been broadly used in software development projects. Tools such as Chianti [117] help developers know what acceptance tests need to be repeated due to changes in the source code. These tools allow team members to discover what source code files could be affected if an acceptance test fails. The current work is different from the traceability tools from two angles. Traceability tools look for change resulting from the source code only, whereas this work additionally takes into account change resulting from unit testing, acceptance testing and continuous integration. Likewise, traceability tools do not consider identifying and co-ordinating the effect of change on development progress, unlike this work.

3.5 Summary

This chapter has discussed managing progress of agile software development. Practices such as *Sit Together* and *Informative Workspace* can facilitate sharing progress information among team members in co-located agile projects. However, when the project is distributed, team members find it harder to maintain an awareness of progress of their tasks.

After discussing distributed software environments, including the motivation for implementing distributed software development environments and the co-ordination challenge in such environments, the chapter reviewed in detail the primary approaches used to manage development progress in distributed agile projects. These are:

- Informal methods: in this approach, *ad hoc* co-ordination mechanisms are used between team members to manage development progress. The main methods include synchronous communication, asynchronous communication, daily tracker, information radiators, and cross-location

visits. The main limitation of these methods is that team members may not recognise the impact of a change on development progress.

- Formal methods: in this approach, the distributed teams use computer systems to keep track of progress information and to manage them. The main methods include Wikis and spreadsheets, traditional project management tools, and agile project management (APM) tools. All these methods were reviewed with a focus on APM tools. A review of 30 APM tools revealed several mechanisms available to assist in supporting the management of distributed agile development. The main limitation of this approach is that the computer systems do not discover the impact of the change on progress but rely completely on team members to recognise it.

The research presented here aims to overcome the limitations of these two approaches through providing a computer-based holistic approach to developing a progress tracking system. This will require identifying the co-ordination support required for managing development progress (Chapter 4) and designing a computer-based system capable of providing the necessary co-ordination (Chapter 5).

CHAPTER 4

Co-ordination Support Required for Managing Progress of Distributed Agile Projects

The goal of this chapter is to identify the co-ordination support requirements for managing the progress of distributed agile projects. The chapter starts by introducing the concept of co-ordination. The main types of co-ordination activity required for managing the progress are then identified. Section 4.3 analyses the progress change events that may result from performing each technical activity, and also provides explicit identification of the co-ordination support required to deal with these events. Two examples are provided in section 4.4 to enhance the understanding of the co-ordination support required for managing the development progress of distributed agile teams. Finally, a short summary for the chapter is given.

4.1 Understanding Co-ordination

Co-ordination is an integral part of teamwork. Mintzberg [118] states:

"*Every organized human activity – from the making of pottery to the placing of a man on the moon – gives rise to two fundamental and opposing requirements: the **division of labour** into various tasks to be performed and the **coordination** of those tasks to accomplish the activity*."

Within the context of software development projects, Mintzberg's observation illustrates that as long as the development process is broken down into tasks and processes, there is a co-ordination requirement [119].

A dictionary definition of co-ordination is 'the act of working together harmoniously' [120]. The definition provides a 'common sense' meaning of the concept. The definition can be divided to three parts:

- 'act': implies that there are *actors*,
- 'working' indicates that actors must carry out *activities*;
- 'harmoniously' implies that the actors perform the activities in order to achieve *goals*.

Hence, actors, activities and goals comprise the main components of co-ordination [121]. Applying the definition to the software development domain, *team members* (e.g. developers, testers) work on *software development activities* (e.g. coding, testing) in order to achieve *the goal of completing the software* requested by the customer.

The dictionary definition is a broad definition; researchers have developed several definitions and theories to understand co-ordination in a more restricted (narrow) way [122]. Chandler defines co-ordination as "structuring and facilitating transactions between interdependent components" [123]. Thompson defines it as "the protocols, tasks and decision-making mechanisms designed to achieve concerted actions between interdependent units" [124]; the National Science Foundation defines it as "the emergent behaviour of collections of individuals whose actions are based on complex decision processes" [125]; Curtis defines it as: "activities required to maintain consistency within a work product or to manage dependencies within the workflow" [126]; Singh defines it as: "the integration and harmonious adjustment of individual work efforts towards the accomplishment of a larger goal" [127].

Some of the above definitions focus on the dependencies between individuals and units while some are concerned with the outcome of the co-ordination [119].

Malone and Crowston developed a general co-ordination theory by the recognition of commonalities in co-ordination problems that were previously considered separately in many different fields, such as economics, computer science, sociology, social psychology, linguistics, organisational theory and management information systems [122].

A co-ordination definition is provided by Malone and Crowston [128] as:

"*Co-ordination is managing dependencies between activities*."

Malone and Crowston see dependencies as dependencies between tasks rather than individuals or units. In addition, their definition concentrates on the case for a need to co-ordinate rather than on the desired outcome of co-ordination. This provides a theoretical framework for analysing co-ordination in complex processes [129].

In a further work [130], they, with other colleagues, characterise the co-ordination dependencies as specialisations or combinations of three basic types of dependencies among activities: flow, sharing and fit. These three types are illustrated in Figure 4-1.

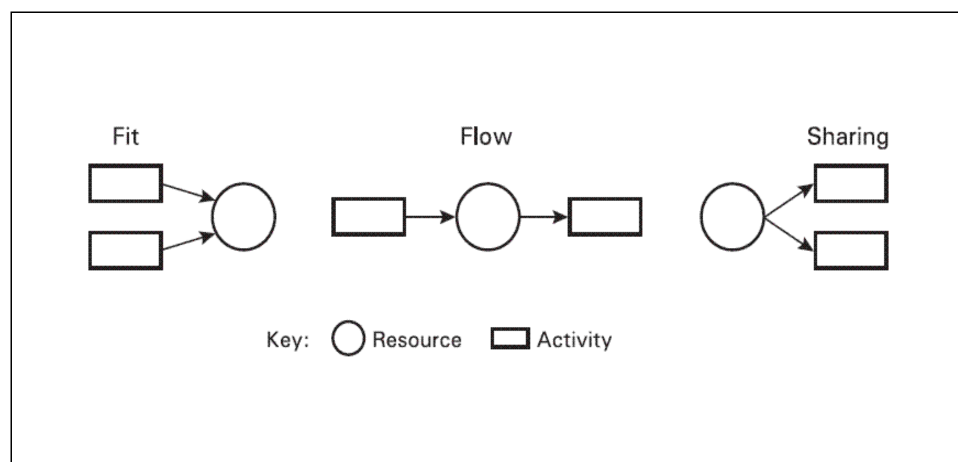


Figure 4-1. The basic types of co-ordination dependencies [130].

- *Flow Dependencies*: arise whenever one activity produces a resource that is used by another activity.

- *Sharing Dependencies*: occur whenever multiple activities all use the same resource.

- *Fit Dependencies*: arise when multiple activities collectively produce a single resource.

Managing progress in agile development requires activities to manage dependencies from all the three basic types of dependency:

- **Flow dependencies:** The agile software development involves a number of sequential activities each of which contributes to making progress towards achieving the goal of completing the software. Team members often need artefacts produced at one stage of the development process in order to perform activities in subsequent stages. An example of this is the acceptance testing for a user story. It cannot be started until the functionalities required for the story are completed.

- **Fit dependencies:** The agile software development involves fit dependencies since a set of tasks can contribute to complete a user story. Moreover, all the user stories performed by the team members contribute to develop the same product. One of the XP practices is to perform continuous integration with the source code produced by the user stories. Assuming that the source code artefacts produced by the user stories US1, US2 and US3 have been integrated by an integration process, any later change to the integrated artefacts should ensure that the interfaces remain consistent.

- **Sharing dependencies:** The agile project is divided into iterations where each iteration produces a releasable version of the system. This means that the user stories are developed within a shared period of time. In addition, a number of different tasks may share the reading of the

same source code artefact. Changes to the shared artefact have to be managed to ensure that the changes made by one team member do not conflict with tasks performed by other team members.

4.2 Types of Co-ordination Activities Required for Managing Development Progress

Performing any of the technical activities affecting development progress, described in Chapter 2 (Table 2-1), will require performing further co-ordination activities to manage change in development progress. The key types of co-ordination activity are: checking progress constraints, identifying potential sources of progress change, reflecting progress change in the tracking system, and finding and notifying team members affected by potential progress change. These co-ordination activities are discussed below.

- **Checking Progress Constraints**

  Indicating progress of tasks, user story and releases requires some conditions that should be satisfied first. Source code artefacts associated with the task must be unit-tested before a developer can register the task as 'complete' in the tracking system. The user story must be integrated and acceptance-tested before it can be described as 'complete'. Releasing is only for the complete stories. Any attempt to violate these conditions needs to be prevented and clarified to the team members.

  Tools such as the versioning system, Team Foundation Server (TFS) [131], enable teams to set policies that enforce every check-in to TFS have an associated unit test written for the code being checked in. TFS offers such policies to improve code quality. Although this policy is not

offered as part of a progress management system, it can be seen as a progress constraint check.

Current progress tracking systems do not apply progress constraint checking for the tasks, user stories or releases. Thus, manual verification is conducted by team members. The manual approach has its limitations. Developers may forget to follow the practices and rules. In addition, with distributed teams, it is more likely that the team members will be unaware of the different technical factors that contribute to violation of the constraints. Code change by other team members may change the development progress because it is not tested or not integrated.

This can result in code with low quality or delay the project because registering a particular user story in the tracking system as ‘active’ and giving a percentage of completion do not reflect the actual *working software* of that user story. In addition, registering it as ‘complete’ does not guarantee that all the unit tests, acceptance tests, integration tests and the required builds have been successfully completed.

- **Identifying Potential Sources of Progress Change**

Generally speaking, the sooner problems affecting progress are discovered, the more likely they can be resolved in the current iteration. When a test fails, team members may spend a significant amount of time identifying potential sources of defects. This is because they may not discover problems until acceptance testing is made. Between making two acceptance tests, a large number of changes which may introduce defects can be performed by team members. It is commonly believed that the earlier a defect is found the cheaper it is to fix it.

The difficulty involved in identification of the source of progress change varies depending on the progress change event taking place. There may be little effort from team members, such as when a tester repeats an acceptance test to complete a user story. If the status of the acceptance test changes from 'pass' to 'fail', this simply means that the corresponding story's progress has been affected. However, identifying the source of the change can be one of the most difficult processes that team members can face. An example of this is when two developers at different geographical locations working on two different tasks use the same source code artefact. One of the developers may modify the source code in a way that affects the progress of the other. This progress change event is hard to track down as the developer affected by the change is not the one who introduced the problem and the originator of the change may be unaware that he has caused the problem.

- **Reflecting Progress Change in the Tracking System**

If the development progress is affected by one of the technical factors described earlier, the impact has to be reflected in the tracking system. For instance, if an acceptance test that is associated with complete story fails, this may lead to changes in the story progress. Such a change has to be reflected in the tracking system. This is important not only because it shows the real progress position of the user stories, but it also shows that the story may need to be modified, re-integrated and undergo acceptance testing again.

Current tracking systems do not provide automatic reflection of the progress of the tasks, user stories and releases. The developer has to change the progress himself. This implies a time overhead; there are lots of daily updates resulting from performing the technical activities.

In addition, most required reflections cannot be easily recognised by the developers. A developer who changes some code may not understand how this change could influence the progress of other tasks.

- **Finding and Notifying Team Members Affected by a Potential Progress Change**

Team members may perform some of the technical activities that affect project progress. It is important that every team member who is affected by a progress change is notified. If a team member is working on a task dependent on another task, he needs to be notified about any progress change to the preceding one.

Current tracking systems do not provide such co-ordination mechanisms. It has to be done manually by team members. This may cause a time overhead due to the frequency of such events. In addition, senders of information do not know the information needs of everyone in the organisation, so they cannot always determine who should receive the information they send [132]. If all team members are informed about all the progress change events, it could result in information overload, so team members may face difficulties in finding the relevant notifications. In other cases there is complexity in understanding the impact of performing the technical activities described earlier, while the notifications may go to team members who are not interested whereas the team members affected by the progress change are not informed.

The lack of mechanisms to identify and notify the right people could lead to serious problems because it can be a reason for delaying the project progress. If a change has been made to a source code artefact that belongs to a complete task or a complete user story and the affected developers are not notified, team members may need to spend a

significant amount of time conducting additional work to resolve issues that occur from the impact of the change. They may not realise that they need to do this, however, until a late stage of the iteration or the project.

4.3 Analysis of Co-ordination Requirements for the Technical Activities

Every technical activity may be carried out in a way that causes a change in development progress. Progress change events need to be recognised as well as the provision of the necessary co-ordination activities (i.e. derived from the co-ordination types discussed in the previous section) that can help managing these events.

This section analyses the progress change events caused by each technical activity and identifies explicitly the co-ordination support required to manage them. It also discusses the distribution effect of co-ordinating technical activities.

4.3.1 Source Code Versioning

Activities involved in source code versioning (create an artefact, update an artefact and delete an artefact) may cause several progress change events that need co-ordination support.

In the case of creating a source code artefact, if the state of corresponding task/story is 'un-started' or 'complete', creating the new artefact for the task/story implies that its state is changed to 'active'. The developer who tries to create the new artefact has to be informed that the task/story is inactive. The state of the task/story has to be changed to 'active' in the progress tracking system. If other team members are affected by the recent task's/story's state, they must be notified.

Updating a source code artefact normally requires that the developer checks out the source code artefact, makes the modification, and then checks it in again. Progress changes resulting from the check-out process are similar to the case of creating a source code artefact.

In case of having a source code artefact shared between two tasks/stories or more, where the state of one of them is 'complete', making check-in to the artefact may change the progress state of that task/story. In this case, which tasks/stories have been affected must be identified. The affected team member must also be found and notified.

If an integrated artefact is modified, it will need to be re-integrated. The recent changes may affect progress of other tasks/stories that share the same artefact. Developers who share a previously integrated artefact to complete their tasks/stories should be made aware that it has new version and, therefore, the artefact need to be re-integrated.

Deleting an integrated artefact may break the build leading to a negative impact on the progress state of a large number of tasks/stories. Affected user stories may need to undergo AT again. If deleting an artefact breaks the build, this needs to be clarified to the developer and deletion may be delayed until the developer discusses the activity with the affected developers. It is important to identify the impact of deleting the artefact on progress and reflect it in the tracking system. Finding and notifying affected team members are also required.

Identifying and co-ordinating the progress change events resulting from the source code versioning are likely to be more difficult if the agile project is distributed. The relationship between the source code artefacts and tasks/stories is difficult to realise with the distributed sites. Consequently, it is difficult to maintain the impact on tasks/stories progress of creating, updating or deleting a source code artefact. In addition, locating and notifying the affected team members may become a significant hindrance. In the case of deleting a source

code artefact, it may be difficult to realise how important the artefact is to team members located at different sites.

A summary of the progress change events resulting from the source code versioning activities and their co-ordination requirements is provided in Table 4-1.

Versioning Activities	Progress Change Event	Co-ordination Requirements	Distribution Effect
Create a new artefact	- Creating a new artefact whose task is ‘un-started’ or ‘completed’ changes the task’s state. - Creating a new artefact whose story is ‘un-started’ or ‘completed’ changes the story’s state.	- Changing state of the task/story if its current state is ‘un-started’ or ‘complete’. - Finding and notifying team members affected may be required.	- The relationship between the artefacts and tasks/stories in the distributed sites is difficult to realise. - It is harder to determine the team members affected.
Update an artefact	- Checking-out a new artefact whose task is ‘un-started’ or ‘completed’ changes the task’s state. - Checking-out a new artefact whose story is ‘un-started’ or ‘completed’ changes the story’s state. - Modifying an artefact whose task is ‘un-started’ or ‘completed’ changes the task’s state. - Modifying an artefact whose story is ‘un-started’ or ‘completed’ changes the story’s state. - Modifying an integrated artefact may require it to be re-integrated.	- Checking-out may require changing the state of the task/story if its current state is ‘un-started’ or ‘complete’. - Sharing new artefact versions should be prevented if corresponding unit tests have failed. - Developers who use a previously integrated artefact should be aware that it has new versions updated and, therefore, the artefact need to be re-integrated. - Finding and notifying team members affected may be required.	- It is harder to realise the impact of updating an artefact on development progress. - It is harder to determine the team members affected.

Versioning Activities	Progress Change Event	Co-ordination Requirements	Distribution Effect
Delete an artefact	- Deleting an integrated artefact may break the build.	- If deleting an artefact breaks the build, this needs to be clarified with the developer and deletion may be delayed until the developer discusses the activity with other developers affected. - Finding and notifying team members affected may be required.	- It is difficult to realise the importance of the artefact to team members. - It is harder to determine who needs to be notified.

Table 4-1. Progress change events, distribution effect, and co-ordination requirements for the source code versioning activities.

4.3.2 Continuous Integration and Releasing

Integration and releasing activities can lead to positive/negative progress change. If an integration process has been performed that failed, team members may not realise which user stories have been negatively affected. The ‘failed’ result should not affect those stories that do not have new versions entered in the build.

An integration ‘pass’ result should contribute to making progress on the affected stories. When a successful integration is made, story owners and testers may not know exactly which stories are ready for the acceptance testing stage. If all the functionalities for a story have been completed and integrated, the tester responsible for the story has to be located and notified that the story is now ready for acceptance testing.

Another potential progress change event may result from the releasing process. A set of user stories may be released while some of them have not been fully tested. In this case, the release process should be prevented. The person making the release has to be made aware that releasing should be for complete stories only.

Continuous integration and releasing activities causing progress change can be difficult to identify and co-ordinate in distributed environments. It is harder to maintain awareness of the effect of an integration result on development progress if the team is distributed. In addition, team members making a release at one site may not know the actual progress state of user stories carried out at another site.

A summary of the progress change events resulting from the continuous integration and releasing activities and their co-ordination requirements is provided in Table 4-2.

CI/Releasing Activities	Progress Change Event	Co-ordination Requirements	Distribution Effect
Perform integration	- A failed integration process has been performed. - A successful integration process has been performed.	- Determining which stories have been affected and reflecting that in the tracking system are required. - When a successful integration is made, testers may not know exactly which stories are ready for the AT. Finding and notifying affected team members may be required.	- It is harder to maintain awareness of the effect of integration on development progress if the team is distributed.
Make a release	- A set of user stories may be released while some of them have not been fully tested.	- A release has to be made for complete stories only.	- Team members making a release at one site may not know the actual progress state of stories performed at another site.

Table 4-2. Progress change events, distribution effect, and co-ordination requirements for the continuous integration and releasing activities.

4.3.3 Unit Testing

Activities involved in unit testing include adding, modifying, deleting and running a unit test. These activities may cause several progress change events that need co-ordination support.

Adding or modifying a unit test without re-testing it or with a ‘fail’ result can affect the corresponding task, if it was complete. It is important to clarify to the developer that state of completed tasks may change due to his activity. It would be safe to prevent the addition or the modification until the unit test passes.

Deleting the only unit test for an artefact of a completed task affects the task’s progress. If it is the only unit test for the corresponding source code artefact, and if the corresponding task is complete, the task state may be affected. It is required to prevent the deletion.

Furthermore, a unit test may not have passed when its corresponding source code version is checked in. In this case, it may affect development progress because getting the unit tests passed is a condition of completing the source code artefacts developed to fulfil requirements of a task. Sharing new source code versions should be prevented if the corresponding unit tests have failed.

Similarly, a unit test may not have passed when a developer wants to set its corresponding task to ‘complete’. It is required then to prevent setting the task to ‘complete’ until all its source code artefacts are successfully unit-tested.

A developer working on his machine may easily understand how adding, modifying, deleting or running a unit test may affect the progress state of the task he is currently working on. The impact of these activities affect the developer who created them only as long as the corresponding source code artefact is not yet shared. However, if the source code artefact is shared with other developers at a different site, it can be difficult to understand the impact on them.

A summary of the progress change events resulting from the unit testing activities and their co-ordination requirements is provided in Table 4-3.

Unit Testing Activities	Progress Change Event	Co-ordination Requirements	Distribution Effect
Add or modify a unit test	- Adding or modifying unit test without testing it or with a ‘fail’ result can affect the corresponding tasks if they were complete.	- It is required to delay the addition/ modification until the test passes.	- Developers may not know which tasks are associated with the unit tests.
Delete a unit test	- Deleting the only unit test for an artefact of a completed task affects its progress.	- Deletion may need to be prevented and the impact clarified to the developer.	- Affected developers may not know the impact of deleting a unit test.
Run a unit test	- A unit test may not have passed when its corresponding source code version is checked-in. - A failed unit test prevents completing the task.	- Sharing new artefact versions should be prevented if corresponding unit tests have failed. - If a unit test fails, the corresponding task must not be set as ‘complete’.	- It will be difficult to realise the impact if the corresponding source code artefact is shared with developers at a different site.

Table 4-3. Progress change events, distribution effect, and co-ordination requirements for the unit testing activities.

4.3.4 Acceptance Testing

Activities resulting from manual and automated acceptance testing (AT) can cause several progress change events that need co-ordination support. These progress change events are discussed below.

Adding or modifying an acceptance test without testing it, or with a 'fail' result, can affect the corresponding story if it was complete. States of the corresponding completed stories may need to be changed. Finding and notifying the owner of the completed story may be required.

Deleting the only acceptance test for a complete user story affects the story's progress. If it is the only acceptance test for the user story, and if the story is complete, the story's state may need to be changed. The story owner and the affected tester must also be found and notified.

Running automated AT may result in two progress change events. First, a complete user story will be affected if one of its associated automated acceptance tests has failed. If an acceptance test fails, the corresponding user story must not be set as 'complete'. Team members affected must also be found and notified.

Second, a user story may be affected if one of its associated automated acceptance tests has passed. This happens if all the functionalities required for the story have been completed and integrated and the other acceptance tests for the same story have already passed. The corresponding user story must be set as 'complete' and team members affected must be found and notified.

Similar to running automated AT, updating a manual acceptance test to 'fail' causes a complete story to become incomplete. In this event, the user story must not be set as 'complete'. Finding and notifying the story owner and the affected tester may be required. Likewise, updating a manual acceptance test to 'pass' may cause the story to become complete The corresponding user story must be set as 'complete' and team members who are affected must be found and notified.

Finally, an acceptance test may not have passed when a team member wants to set its corresponding story to 'complete'. It is required then to prevent setting the story to 'complete' until all its corresponding acceptance tests pass.

With distributed agile projects, it is possible to have testers and story owners scattered at different sites. Adding, modifying deleting or running an acceptance test may affect its corresponding story. Testers may encounter difficulties in finding and targeting notifications to those story owners affected if they are at different sites. In addition, running automated acceptance tests frequently as part of a build process may result in having a large number of progress changes to the stories they belong to. It can be difficult to manually find and notify team members affected as size of the acceptance test grows.

A summary of the progress change events resulting from the acceptance testing activities and their co-ordination requirements is provided in Table 4-4.

Acceptance Testing Activities	Progress Change Event	Co-ordination Requirements	Distribution Effect
Add or modify an acceptance test	- Adding an acceptance test without testing it or with 'fail' result can affect the corresponding story if it was complete. - Modifying an acceptance test without testing it or with 'fail' result can affect the corresponding story if it was complete.	- States of corresponding completed stories may need to be changed. - Finding and notifying affected team members may be required.	- Team members may not know impact of adding an acceptance test.
Delete an acceptance test	- Deleting the only acceptance test for a completed story affects its progress.	- Story state may need to be changed. - Finding and notifying affected team members may be required.	- Developers may not know the impact of deleting an acceptance test.

Acceptance Testing Activities	Progress Change Event	Co-ordination Requirements	Distribution Effect
Run an acceptance test	- A complete user story will be affected if one of its associated automated ATs has failed. - A user story may be affected if one of its associated automated ATs has passed. - Updating a manual acceptance test to 'fail' may cause a complete story to become incomplete. - A failed acceptance test prevents completing its corresponding story.	- If an acceptance test fails, the corresponding story must not be set as 'complete'. - State of the relevant story may need to be changed. - Finding and notifying affected team members may be required.	- Testers may encounter difficulties in finding and targeting notifications to those affected.

Table 4-4. Progress change events, distribution effect, and co-ordination requirements for the acceptance testing activities.

4.4 Examples

The last section identified the co-ordination requirements for each progress change event. This section shows examples of the sequence of co-ordination steps that each technical activity has to get through. This can cover enhanced understanding of the co-ordination support required to manage the development progress of distributed agile teams. Two examples are provided. An update activity to a source code artefact normally requires two smaller activities: check-out and then check-in; the co-ordination support required for these activities is illustrated.

A typical **check-out process** will involve the following steps:

- Before checking-out a source code artefact, *it must be ensured that the corresponding task and story are still active*, because developers can only work on an active task/story.
- If the task and/or story are active, the developer can check-out the artefact. Otherwise, *the progress state of the corresponding task and/or story must be changed*. This needs to be explicitly shown on the tracking system so that the whole team will be aware of the actual progress of the project.
- *The team members affected must be found and notified:* team members affected (i.e. story owner/tester) may be at different sites. It is important to look for team members affected and notify them in order to resolve problems as early as possible.

The co-ordination required for the check-out process is described in the following UML activity diagram (Figure 4-2):

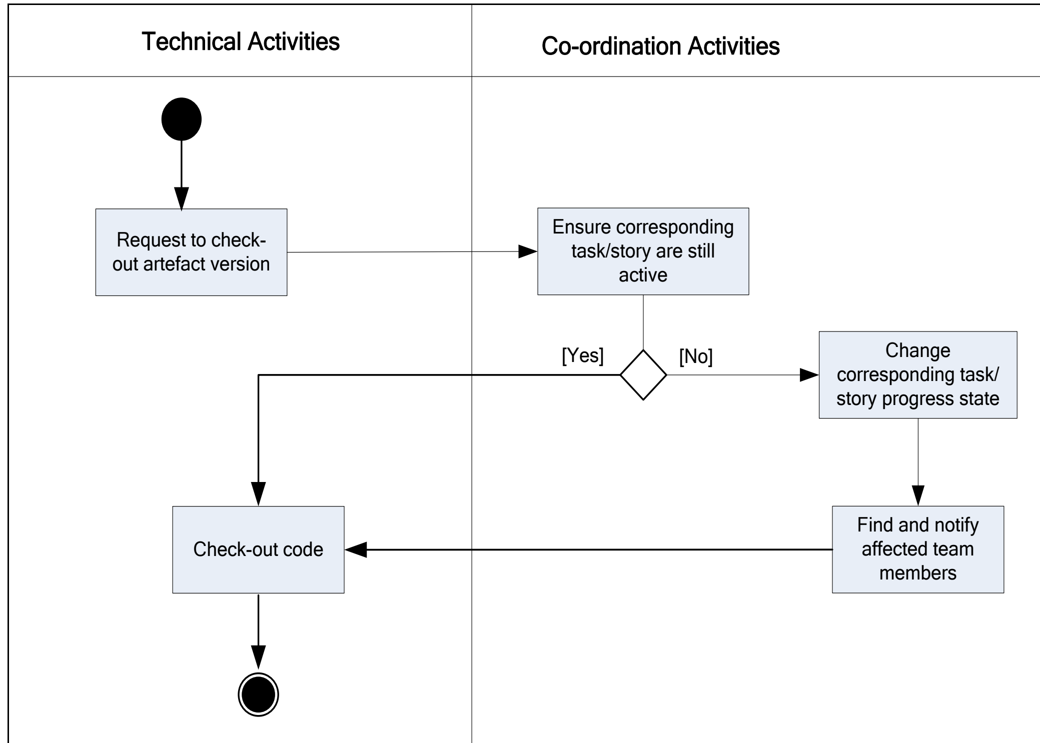


Figure 4-2. Co-ordination support required for the check-out process.

A typical **check-in process** will involve the following steps:

- Before checking-in the source code artefact, the developer has to *ensure all corresponding unit tests are successful.* Only unit-tested code can be shared with other team members.
- If one or more of the unit tests has failed, the developer has to refactor the code and try testing again. If all the unit tests are successful, the developer can check-in the artefact. If the recent change affects the progress of other stories, *the affected stories must be identified.*
- *The progress state of the stories affected must be changed.*
- Finally, *the team members affected must be found and notified.*

The co-ordination required for the check-in process is described in the following UML activity diagram (Figure 4-3):

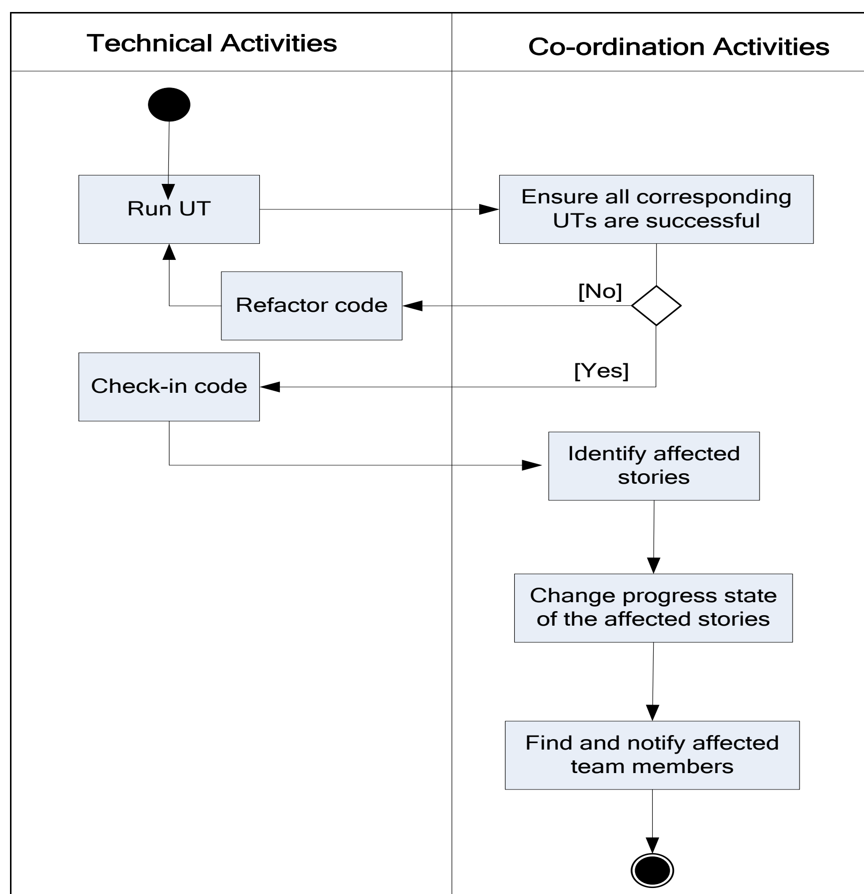


Figure 4-3. Co-ordination support required for the check-in process.

4.5 Summary

This chapter has discussed the concept of co-ordination. It has shown how different researchers have different opinions on defining co-ordination. More emphasis has been given to the co-ordination theory established by Malone and Crowston. One of the important contributions of the coordination theory is its definition that looks at co-ordination as managing dependencies between activities. The definition focuses attention on the causes for co-ordination, which can help analyse co-ordination in complex processes. Section 4.1 argued that managing progress in agile development requires activities to manage the three basic types of dependencies identified by Malone and Crowston (i.e. flow, fit and shared dependencies).

Section 4.2 identified four key types of co-ordination activity for managing progress of distributed agile projects. These are:

- Checking progress constraints.
- Identifying potential sources of progress change.
- Reflecting progress change in the tracking system.
- Finding and notifying team members affected by potential progress change.

Technical activities may cause progress change events that require performing further co-ordination activities, derived from the four co-ordination types identified above. Section 4.3 identified explicitly the co-ordination requirements associated with each progress change event. The check-out and check-in examples were provided in section 4.4 to illustrate how these activities may cause progress change events and what co-ordination support is required to manage these events.

Current formal methods discussed in Chapter 3 do not support co-ordinating progress changes resulting from these technical activities; hence, co-ordination activities are performed informally. Due to limitations of informal methods, a

computer-based system that takes into account the various technical activities affecting progress is essential. The next chapter discusses how such a system can be designed.

CHAPTER 5

Design of Progress Tracking System

This chapter discusses the design of a computer-based progress tracking system capable of providing the necessary co-ordination requirements identified in the previous chapter. It starts by describing an architecture that enables the tracking system to recognise the impact of the technical activities on development progress. Since judging progress in agile development is primarily based on the source code state, a version model has been developed to identify the level of maturity of each source code version (section 5.2). In addition, to provide team members with better awareness of the progress of user stories, a novel user story progress model has been proposed (section 5.3). The process model and the data model for the progress tracking system are provided in sections 5.4 and 5.5 respectively. Finally, design issues are discussed in section 5.6 before summarising the chapter in section 5.7.

5.1 System Architecture

There are several technical activities affecting development progress as discussed in Chapter 2. These activities are currently carried out by technical systems:

- Source Code Versioning: carried out by version control systems
- Unit Testing (UT) activities: carried out by UT tools
- Acceptance Testing (AT) activities: partially carried out by AT tools

- Continuous Integration (CI) and Releasing activities: carried out by CI and releasing tools.

The problem with these systems is that they work separately from the progress tracking system. Hence, if a progress change event is caused by a technical activity, the progress tracking system cannot identify it.

The progress tracking system proposed here enables the tracking system to keep track of the impact of the technical activities by placing them under control of the tracking system (Figure 5-1). This can be achieved by:

- Integrating the versioning functionalities into the progress tracking system,
- Linking the UT tool, AT tool and CI tool with the progress tracking system.

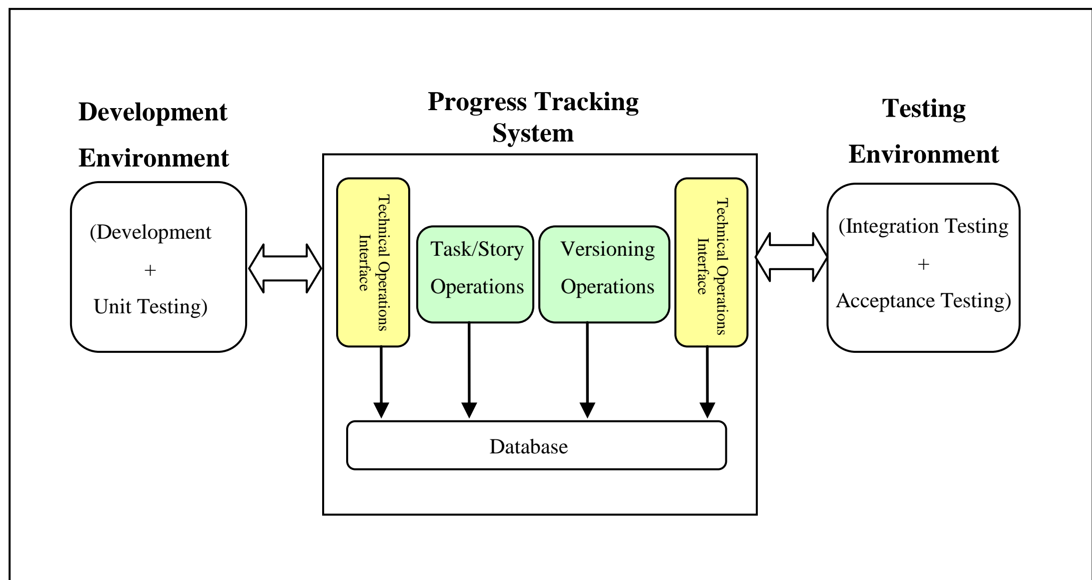


Figure 5-1. A High level architecture for the progress tracking system.

- **Integrating Versioning Functionalities into the Tracking System**

  Current versioning systems provide technical mechanisms to store and control source code artefacts. However, these systems provide no support for identifying and co-ordinating changes affecting development progress.

  Because development progress in an agile development is directly based on the maturity of the source code artefacts, tasks/stories should not be tracked separately from the source code artefacts that determine their functionalities. There should be a consistency between the progress data and the actual work performed by developers. This has been seen as a worthy reason to fully integrate versioning functionalities into the progress tracking system.

- **Linking the UT, AT and CI Tools with the Tracking System**

  The progress tracking system has to offer interfaces to the UT tool, AT tool and CI tool, so the tracking system can capture the point where a potential progress change takes place.

5.2 Version Model

5.2.1 Version States

Version states are used to indicate the level of maturity of different versions of source code artefacts. Version state is taken into account when determining progress. Based on the fact that source code artefacts pass several stages before

they are released (unit testing, integration, releasing), a four-stage hierarchical promotion model that shows this evolution is proposed which incorporates the following versions:

- Transient Version (TV): the artefact version is not shared with other team members.
- Unit-Tested Version (UTV): the artefact version is unit-tested and available to be shared with other team members. The artefacts in the unit-tested stage are prepared for the next integration so this stage can be seen as the 'Ready-for-Integration' stage.
- Integrated Version (IV): the artefact version is unit-tested and has passed the build.
- Releasable Version (RV): The user stories for which the artefact version provides functionality have passed AT and are ready for releasing.

The concept of version states in versioning systems is not new. It has been widely applied in versioning systems built to support change management in software design and engineering design (e.g. [133] [134] [135]). However, unlike this work, previous versioning systems do not incorporate 'agile' maturity. The promotion model in these systems does not serve the purpose of supporting agile software projects specifically.

5.2.2 Version Operations

Current versioning systems capture the point where change is instigated but these systems do not show and co-ordinate the impact of change on the agile progress. New operations are required to fulfil the requirements of providing a better description of artefact progress states. Extended versioning operations are described in Table 5-1.

Versioning Operation	Description
Create a new artefact	A new artefact is created as transient version (TV) in a developer's workspace.
Check-out artefact version	A new TV can be created from a version of an existing artefact. The version is created as part of specific task duty.
Check-in artefact version	If TV is stable and unit-tested, it can be promoted to UTV.
Perform integration	If integration is successful, all UTVs included in the integration are promoted to IV.
Release artefact version	If acceptance testing is successful for all affected stories, their associated versions can be released to the customer.
Delete an artefact	An artefact is deleted.

Table 5-1. Extended versioning Operations.

The UML Statechart Diagram in Figure 5-2 shows how the new versioning operations can change an artefact's state.

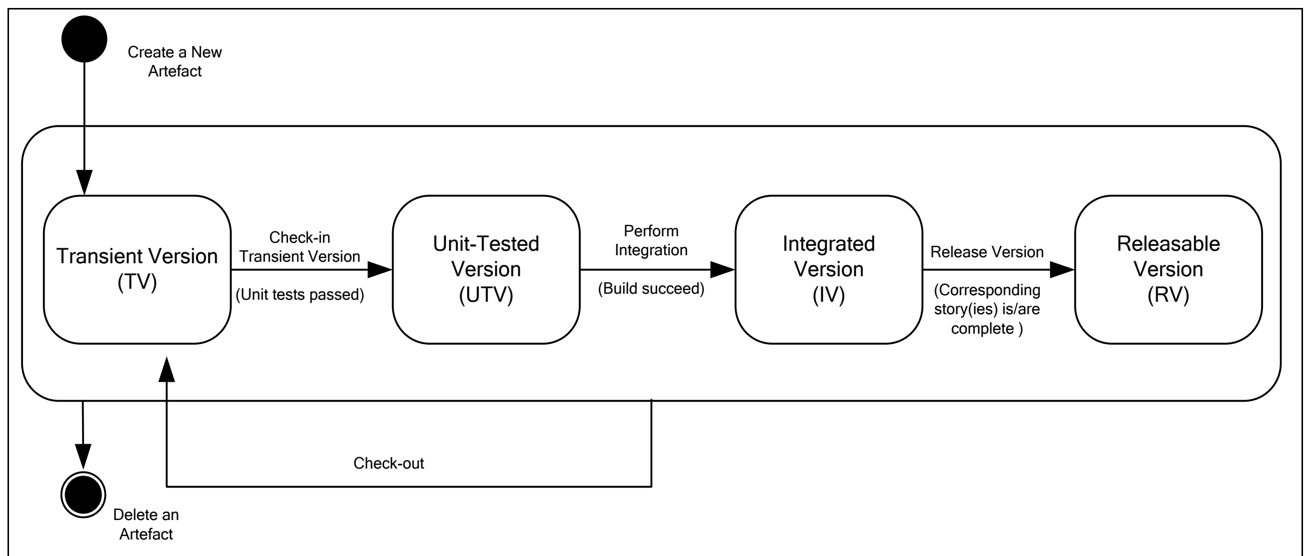


Figure 5-2. Source code version states.

2.2.3 Version Tracking

Managers need to know which tasks are working on each source code artefact. This information can help to recognise which tasks are affected by progress change. In order to obtain this information, every time a developer tries to change an artefact, the task he is working on has to be identified.

A linkage for each source code artefact can be created that describes the tasks that are using versions of the artefact and at which state they are. Table 5-2 shows an example of an artefact with different versions for different tasks.

Version	Task1	Task2	Task3
TV			V3
UTV		V2	
IV	V1		
RV			

Table 5-2. Each source code artefact is linked to the tasks working on it.

The linkage allows the two important versions for each source code artefact to be kept track of. They are the last stable version (last IV), and the last recent version (last UTV). When a developer asks to read or create a new version, he can choose either one. This provides the awareness to the developer about the status of the version he is using.

The promotion to a higher state affects all the tasks/stories that have the same or lower state. The tasks that have TVs are not usually affected by new updates as the TVs represent unstable copies that are isolated from the other developers. Table 5-3 illustrates when a new version can affect current versions. The main driver of producing such a table is to identify clearly the situations in which editing or promoting a source code artefact by one task can have an effect on other tasks' progress.

	Current TV	**Current UTV**	**Current IV**	**Current RV**
New TV	No	No	No	No
New UTV	No	Yes (developers who use old version should be informed)	No	No
New IV	No	Yes (The task that is linked with the UTV should be updated with the new version)	Yes (developers who use old version should be informed)	No
New RV	No	Yes	Yes	Yes

Table 5-3. New version affects current versions.

If a version is promoted to UTV, all the tasks that have UTV versions for the same artefact can be affected. The new modifications for the artefact may influence the work recently completed by other tasks, which are not integrated.

However, if a new UTV is produced, the tasks that have IVs are not affected because the UTV version can be seen as a second transient version due to the fact that it is not integrated yet.

Furthermore, it is important to prevent the new build result from influencing complete stories. Let us assume that a story US1 is complete and new versions of the artefacts that US1 used are produced by another story and have undergone a build. If the build failed, it should not affect the stories that are already complete. The progress impact should apply only to those stories that made recent changes.

5.3 User Story Progress Model

Status of the agile projects is expressed through determining the state of the user stories, where each story should produce a block of working software. In order for the story to produce a working software, the individual source code versions contributing to fulfilling the requirements of a story have to be integrated successfully (i.e. reach IV stage) and the set of versions together has to be acceptance-tested in a testing environment to ensure the customer is satisfied with the story. The version model presented in the previous section helps determine the state of the software (source code) and hence can help describe the state of user stories.

User story states are commonly identified as ‘Un-started’, ‘In Progress’ or ‘Done’ (e.g. [17] [100]). There is often doubt regarding the meaning of the ‘Done’ state. Some agile teams assume that a complete story means that all tasks included in a story are complete. This interpretation does not satisfy the agile definition of the completed stories, which also requires stories to be integrated and acceptance-tested. Sutherland et al. [177] stress the point that user stories must only be considered complete after testing. They pointed out that failure to do this allows work in progress to spread, introducing waiting times and greater risk into the project.

The user story’s state may change, even after performing acceptance testing (AT). Source code artefacts related to a completed story may be versioned and need to be re-integrated and re-tested. To satisfy this, we have identified a new model for the story progress states that takes into account the different progress stages that a user story can assume (Table 5-4). User stories may assume one of the following states: ‘Not started’, ‘Active’, ‘Waiting for integration’, ‘Waiting for AT’, or ‘Complete’.

The user story progress model supports providing a more realistic view of the actual state of the software project and also helps reflect the impact of the technical activities on development progress.

A user story becomes 'Active' once a developer works on one of its corresponding tasks. After implementing all its functionalities, it moves to the 'Waiting for Integration' state. Once the integration is passed, it moves to the 'Waiting for acceptance testing' state. Finally, it can only become complete if all the associated acceptance tests pass. Team members will have better awareness of the progress state of user stories if they can obtain detailed information about these midpoints.

Story State	Description
Not started	User story has not been started yet.
Active	One or more of the story's tasks is still active.
Waiting for integration	All included tasks are complete, but integration has not been conducted yet, or it has failed.
Waiting for acceptance testing	Integration has been successful, but acceptance tests have not yet been performed, or they have failed.
Complete	Integration has been successful and acceptance tests have successfully passed

Table 5-4. User story progress model.

At the beginning of an iteration, all user stories are in the 'Not started' state. However, the technical activities described in Chapter 2 (Table 2-1) may change stories' states and move them from one state to another.

Checking-out a source code artefact version, as part of working on a user story, causes the story to become 'Active'. The completion of implementing all the tasks corresponding to a user story makes the story move to the 'Waiting for integration' state (this implies that source code artefacts associated with each task

have been unit-tested). If integration is passed, the story becomes ready for acceptance testing (moves to the 'Waiting for AT' state). If all acceptance tests associated with the user story have passed, the story then becomes 'Complete'.

Creating a new artefact or checking-out artefact versions means that there is still work needed to fulfil requirements of the story, i.e. the story is still 'Active'. Editing a source code artefact that belongs to a 'Waiting for AT' story or a 'Complete' story may require that the story undergoes integration once again. In addition, deleting a source code artefact belonging to a complete story may require performing further integration. Moreover, adding or modifying an acceptance test for a 'Complete' story moves the story to the 'Waiting for AT' state. Figure 5-3 is a UML state diagram showing how the technical activities may move a user story from one state to another.

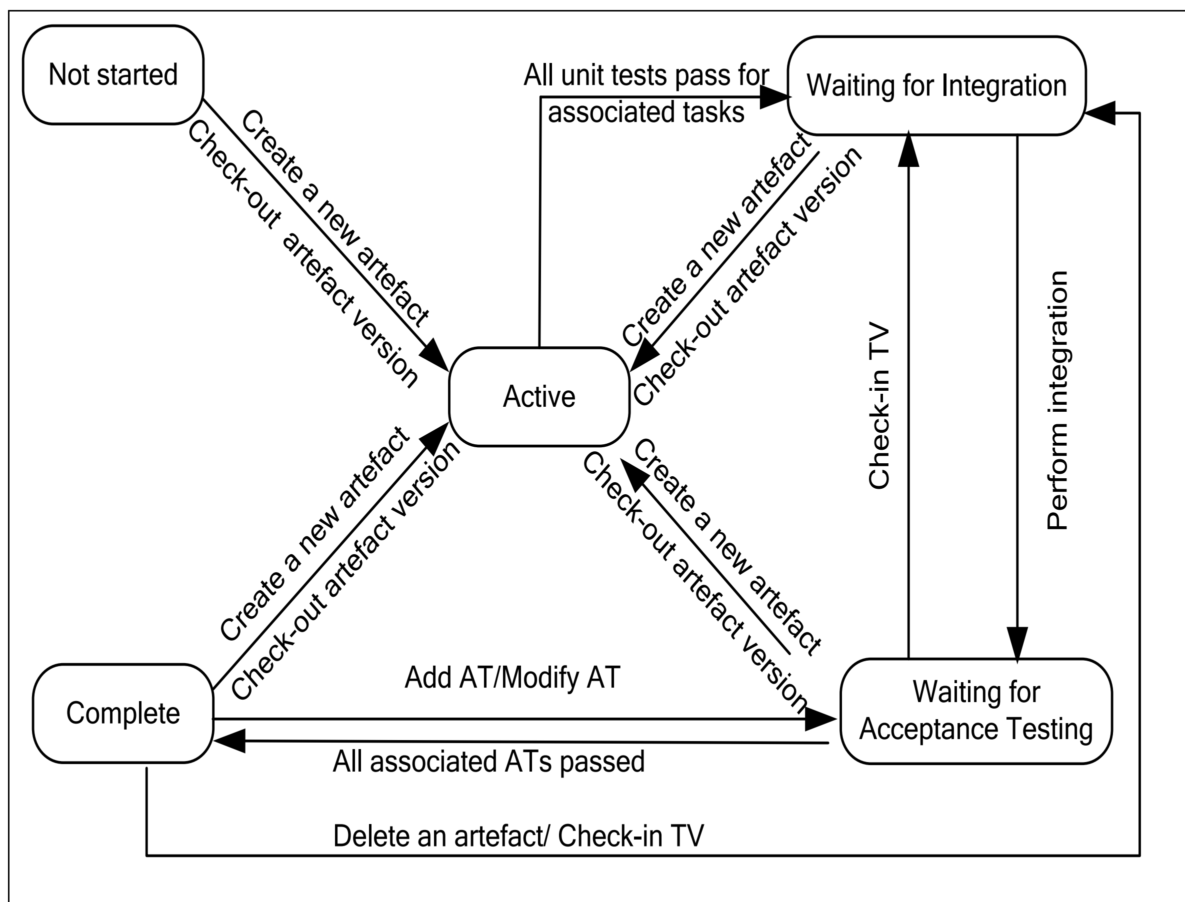


Figure 5-3. A UML story state diagram.

5.4 Process Model

A set of process models needs to be developed to illustrate how each technical activity affects development progress. These models have to provide a visual representation of how the co-ordination activities discussed in Chapter 4 can be implemented in a computer-based system.

5.4.1 Selecting a Process Modelling Technique

The technique chosen to represent process models for the technical operation has to be able to fulfil the following requirements.

- It must show the co-ordination required to manage development progress. A behavioural model must clearly show the sequence of activities in a process. This includes representing sequential and parallel activities as well as the events that trigger activities.
- It should be transparent. Visual representation of each type of technical operation is needed. The value of making the processes transparent is that they can be examined and modified if necessary.
- It should be capable of representing roles in the process. Technical operations involves performance by different types of team member (i.e. developers, story owners, testers) and different technical systems (i.e. progress tracking system, UT tool, AT tool, CI & releasing tool).

Other general requirements necessary for the selected modelling techniques include [136]:

- sufficiently expressive
- easy to use
- unambiguous
- supported by suitable tools
- widely used.

In order to identify appropriate techniques for process modelling, a review of software engineering literature has been conducted [137] [138] [25] [139]. The review revealed the availability of a wide range of options.

Text-based modelling and pseudo-code can provide powerful ways for expressing ideas; however, they can result in a large amount of text, which can be a barrier to reviewing the models properly.

Flow charts can be used to show the flow of control but fail to represent the roles involved in the process. Data Flow Diagrams (DFDs) are easy to communicate with users and provide the flexibility to abstract any level of details. The main limitation of a data flow diagram is that it does not show flow of control. If several outputs may result from decisions within a transformation, a data flow diagram shows only the different possible outputs, not the decisions taken. In addition, a data flow diagram does not model time-dependent behaviour well [140].

UML Activity Diagrams were chosen as the techniques to model the technical activities. They can fulfil all the requirements identified above. They are able to provide behavioural models to clearly represent both sequential and concurrent activities. They also provide transparent processes that explicitly show the co-ordination required to support tracking progress. Moreover, they can represent different roles involved in a process.

Most software developers can easily understand the notation for UML Activity Diagrams. UML has been adopted as the industrial standard for object-oriented modelling by the Object Management Group [141].

5.4.2 Modelling the Technical Processes

A process model is developed for each technical activity (Appendix B). It provides a visual representation of how the co-ordination activities can be implemented. This includes showing explicit support for checking progress constraints, finding and notifying team members affected by progress change,

identifying potential sources of progress change, and reflecting progress change in the tracking system.

As an example, the check-in process model is provided in Figure 5-4. It shows the sequence of activities involved in checking-in a source code artefact version. The process model incorporates the various co-ordination requirements identified for the check-in operation identified in the example of Section 4.3.

- Before checking-in the source code artefact, the developer has to *ensure all corresponding unit tests are successful*. Only unit-tested code can be shared with other team members. If one or more of the unit tests has failed, the developer has to change the code and try testing again.

  The tracking system automatically checks whether there is a unit test associated with the source code artefact. If there is no unit test, the tracking system displays a message to the developer informing him about that.

  If the tracking system discovers that there is a unit test associated with the artefact, a unit testing request is sent to the unit test tool. The tracking system then gets the unit testing result. If it shows that the test failed, the tracking system informs the developer that the check-in process cannot be completed and he needs to ensure that the unit test passes before trying to check-in the code again.

- If all the unit tests are successful, the developer can check-in the artefact. If the recent change affects the progress of other stories, *the affected stories must be identified*.

  If the unit test passed, the source code version is checked-in and its state is updated to 'UTV'. The tracking system identifies the stories potentially affected that are in the 'Waiting for AT' state or in the 'Complete' state. The developer selects the stories that he thinks are likely to be affected.

- *The progress state of the stories affected must be changed.*

  The progress state of the affected stories are changed to ‘Waiting for Integration’.

- Finally, *the team members affected must be found and notified.*

  The owners and testers of the affected stories are found and notified. Other affected team members, such as developers who completed a task that used the source code, need also to be identified and notified.

The models are abstractions which show how the technical processes can be designed. However, they provide only one possible way to model the technical processes. Different agile projects may have different requirements based on their working practices. Therefore, the proposed models can be adapted. For instance, some agile teams may prefer not to use the unit testing check associated with every check-in process. They may prefer to relax this constraint by leaving the unit testing check as a policy option for team members.

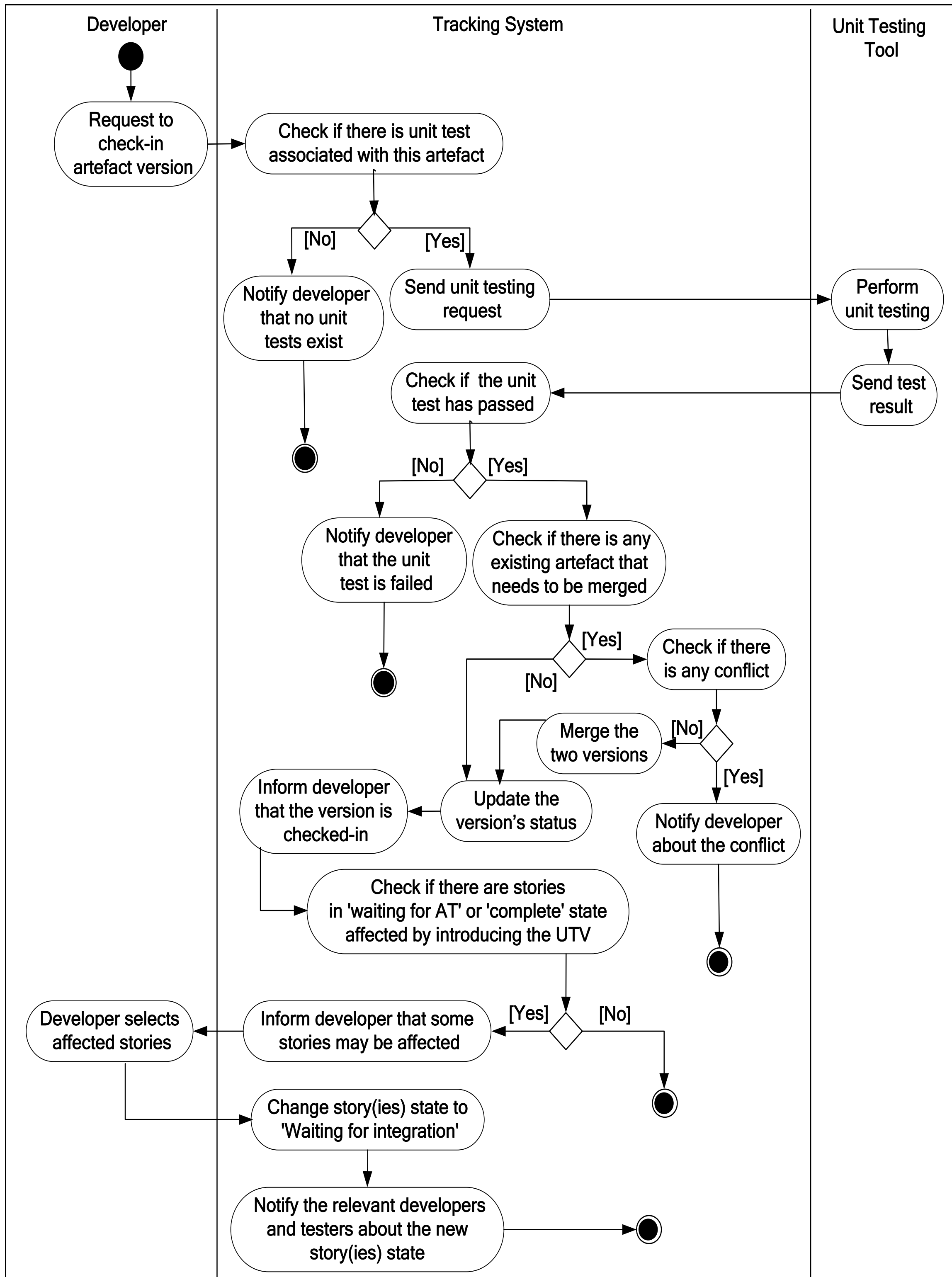


Figure 5-4. The 'Check-in' process model.

5.5 Data Model

The proposed progress tracking system requires storing and accessing different types of data entity. These data entities have dependencies among them. There is a large number of dependencies among tasks, stories, releases, unit tests, acceptance tests and integration tests. These dependencies in the tracking system need to be carefully represented in a data model. The UML Class Diagram in Figure 5-5 shows the relationships among the main entities in the system.

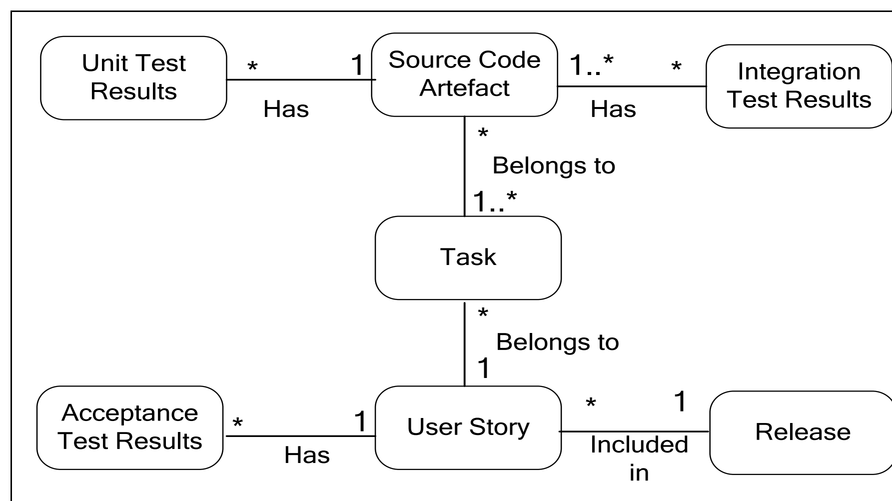


Figure 5-5. UML Class Diagram for the main entities in the tracking system.

The model shows a logical representation for the data in an agile software project. A release consists of user stories, where each story needs at least one acceptance test to test it. A user story consists of tasks and within each task one or more source code (development) artefacts is created. A source code artefact should have at least one unit test and may be included in many integration processes.

The model in Figure 5-5 represents only the main entities in the progress tracking system. Figure 5-6 extends it by representing the team members' information: developer (develops source code artefacts as part of his work on a task), story owner (is responsible for ensuring completing a story as customer requires), tester (is responsible for testing one or more acceptance tests), and project manager

(takes responsibility for managing the whole stories in a release). The model also represents the following data entities:

- **Development Version:** each development artefact may have one or more versions.
- **UT Version:** each unit test artefact may have one or more versions.
- **AT Version:** each acceptance test artefact may have one or more versions.

The data model shows the minimum data requirements needed to design a simple progress tracking system using the holistic approach. A summary of the entities included and their attributes is provided in Table 5-5 below.

Data Entity	Attributes
Release	ID, Planned Start, Planned Complete, Actual Start, Actual Complete
Iteration	ID, Start Date, Complete Date
User Story	ID, Name, State, Planned Start, Planned Complete, Actual Start, Actual Complete
Task	ID, Name, State, Planned Start, Planned Complete, Actual Start, Actual Complete
Development Artefact	ID, Name, Last UT, Last IV, Last RV
Development Version	ID, Time Stamp, Status
UT Artefact	ID, Name
UT Version	ID, Time Stamp, Status
AT Artefact	ID, Description, Type
AT Version	ID, Time Stamp, Status
Developer	ID, Name, Tel, Location
Tester	ID, Name, Tel, Location
Story Owner	ID, Name, Tel, Location
Project Manager	ID, Name, Tel, Location

Table 5-5. Data entities and their attributes.

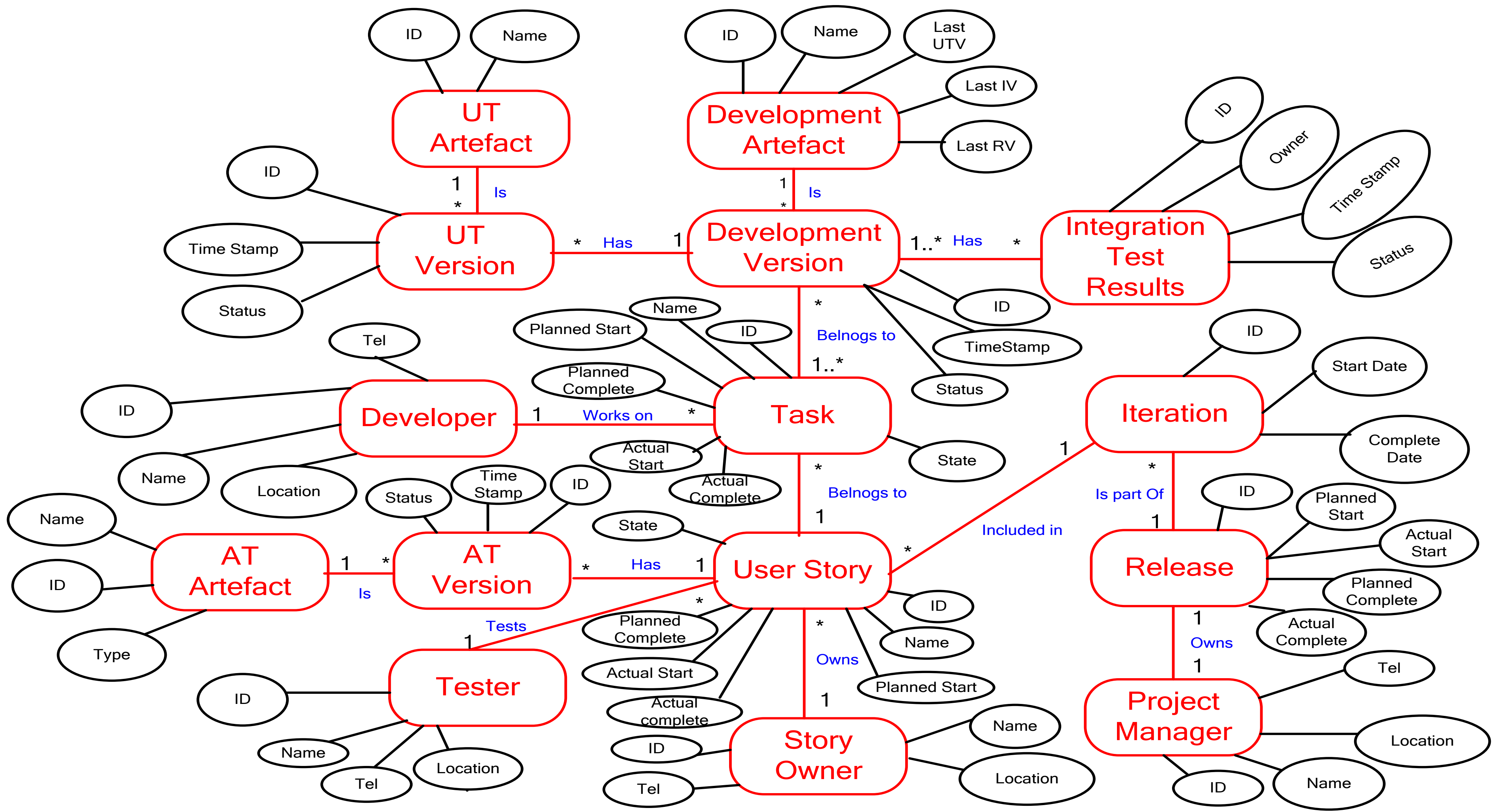


Figure 5-6. Detailed data model.

5.6 Design Issues

There are different techniques and approaches used by agile teams to apply the agile practices. This section discusses the acceptance testing approaches and the continuous integration approaches and which approaches are supported in the proposed progress tracking system.

5.6.1 Acceptance Testing Approaches

During an iteration, user stories will be translated into acceptance tests. Acceptance tests are high level tests of user stories and are used to ensure that the functionalities implemented in stories meet customer requirements, rather than as a means of testing internal or technical elements of the code, as this is done by unit tests.

Acceptance tests can be automated by means of acceptance testing frameworks such as Fitness [142] and Selenium [143]. There is a debate in the agile community regarding the value of automating the acceptance tests. While some proponents like Ron Jeffries[2], believe that the use of an acceptance testing tool is essential to the success of the agile project [144], others believe it can be costly, as well as being time consuming. James Shore[3] states that he no longer uses automated acceptance testing or recommends it [145]. He adds:

"My experience with Fit and other agile acceptance testing tools is that they cost more than they're worth. There's a lot of value in getting concrete examples from real customers and business experts; not so much value in using "natural language" tools like Fit and similar" [145].

---

[2] Ron Jeffries is one of the 3 founders of XP and the author of *'Extreme Programming Installed'* book.

[3] James Shore is a thought leader in the agile community and the author of '*The Art of Agile Development'* book.

According to a survey by VersionOne published in December 2010 [54], more than two-thirds of the agile community do not use any automated AT tools. In addition, there are some tests that cannot be automated and need to be performed manually. If testing is manual, there will usually be a longer interval before the test is performed again because performing tests is often extremely time-consuming. This makes it harder to recognise the source of the failed tests. It also causes team members to rely on less accurate progress information.

In order to provide a general approach in this research, the proposed progress tracking system provides support not only for the manual acceptance testing but also for the automated approach. The proposed architecture allows receiving acceptance testing information from the acceptance testing tool.

Furthermore, additional functionalities are provided to the automated AT process. The tracking system sends an AT request to the AT tool and then receives the result. The tracking system analyses the AT result and finds out how it affects the development progress. If a complete user story is affected due to one of its associated automated acceptance tests failing, the corresponding user story is set as ‘Waiting for AT’. Team members affected are found and notified. In addition, if a user story is affected because one of its associated automated acceptance tests has passed (i.e. all the functionalities required for the story have been completed and integrated and the other acceptance tests for the same story have already passed), then the corresponding user story is set as ‘Complete’. Team members affected are also found and notified.

5.6.2 Continuous Integration Approaches

Continuous integration (CI) is proposed in XP to eliminate the problems of the traditional integration by building the system incrementally. There are two approaches to provide the CI; either synchronous or asynchronous. The

synchronous CI means that every commit to the repository builds the system as Martin Fowler suggests [47].

Developers test their own changes before committing to the repository. If a personal build is done successfully, the developer can check-in code in confidence that his/her code has been tested. If it fails, the developer is free to fix the problem and re-test before committing. The code is built again immediately after the check-in and any failed results between the two builds are notified. GO [146] and Pulse [147] are examples of CI systems that support the personal build strategy.

Pre-commit build is another synchronous strategy. It enforces the build before every check in as in TeamCity [148] and Gauntlet [149]. However, it is not always possible to make a build and integration tests with every commit. Poon [150] observes that:

"With tests that took 3 hours to run, how could we do continuous integration? We were never going to get multiple check-ins per day".

Poon suggests checking-in the artefacts to branches first and then if the tests passed, it is integrated to the shared mainline. This strategy can still be described as synchronous because the artefacts are not shared until a successful build. There are few versioning systems that can support a high number of branches (e.g. plastic SCM [151]) whereas most of them do not have strong merging support. All the previous strategies might prevent the developers from sharing some code that they could need during working on their tasks for a long time.

The synchronous approach has its limitations. The primary disadvantage is the difficulty in implementation without affecting productivity in other ways. Forcing tests in a trigger can be a bottleneck for commits; while a build is running for one commit other commits will be blocked [152]. Besides, it might be too long a time until code is shared.

The asynchronous integration approach allows the team to share code that is either integrated or ready for integration. The integration might be done once or twice a day. This provides a more flexible and more general integration strategy that uses the continuous integration practice and can still be useful in broader situations (i.e. distributed teams). The asynchronous CI strategies have been classified by the author to two main strategies:

The first strategy is to use one mainline that is integrated periodically. The developers share one repository and one mainline. The disadvantage of this strategy is that the developers might use not integrated or possibly not unit-tested artefacts. Examples of versioning systems which support this strategy include Subversion [153] and Clearcase [154].

The second strategy is the multi-stage continuous integration. Developers in each site integrate their work together first before doing a bigger continuous integration between the different sites. An example of SCM that supports this strategy is Accurev [155] .

The mini integrations in each site isolate the integration problems and facilitate identifying the source of the defects. However, although this strategy can solve the limitation of the one mainline by sharing only the mature artefacts between the different teams, it might prevent the team from sharing some code that they could need for a long time (the same existing problem in synchronous CI). The multi-stage continuous integration also requires the dispersed teams to be well decoupled [156].

A summary of the strengths and weaknesses of the discussed approaches and strategies is provided in Table 5-6.

Approach	Strategy	Strengths	Weaknesses	Systems Supporting the Strategy
Synchronous	Personal builds	- Few defects can only be shared.	- Defects could be shared. - Long time before sharing the code.	- GO - Pulse
	Pre-commit build	- Integrated code can only be shared.	- Can be a bottleneck for commits. - Long time before sharing the code.	- TeamCity - Gauntlet
	Private branch	- Integrated code can only be shared.	- Long time before sharing the code.	- Plastic SCM
Asynchronous	One mainline	- Developers do not need to wait long time before seeing others' code.	- Developers can share not integrated code and possibly not unit-tested code. - Developers do not know the last integrated version	- Subversion - Clearcase
	Multi-stage integration	- The integrated code is only shared between the distributed teams. - Facilitate identifying the source of the defects.	- Long time before sharing the code between the sites. - The dispersed teams need to be well decoupled.	- Accurev

Table 5-6. Continuous integration approaches and strategies.

Beck [42] mentioned that the asynchronous approach is the most common style of continuous integration. The weaknesses of the synchronous approach are worse than the weaknesses in the asynchronous approach. Keeping the team too long a time before sharing the code or stopping the commits for the pre-commit build can adversely affect the agility and productivity. Beck also recommends using one mainline strategy. He observes that multiple code streams are an enormous source of waste in software development.

Therefore, for the purpose of this research, the asynchronous approach is more appropriate to be part of the progress tracking system proposed in this research. Furthermore, the one mainline strategy is sufficient to support the research. It can

demonstrate that the integration in the distributed agile development can be managed by the holistic approach. Showing the process in one stage is sufficient because the process support that will be taken in the other stages is similar.

Although there are several integration systems that support the chosen strategy (e.g. Subversion and Clearcase), these systems do not show impact of the integration activity on development progress. The proposed tracking system overcomes this limitation by providing a process that allows team members to know which user stories have been positively/negatively affected due to an integration activity.

5.7 Summary

An approach for designing a progress tracking system for distributed agile teams has been proposed. The system pays attention to the impact of the technical activities on development progress. It keeps track of the impact of the technical activities by placing them under control of the tracking system. This has been achieved through integrating the versioning functionality into the progress tracking system and linking the UT tool, AT tool and CI tool with the progress tracking system.

The chapter has introduced four types of model that serve diversified needs.

- **Version Model:** This four-stage hierarchical promotion model shows the progress of each source code version from the time of the developer creating it until it becomes ready for release. Knowing the current progress state of source code enables agile teams to identify the real progress of a specific task/user story.
- **User Story Progress Model:** A better awareness of the progress state of user stories can be achieved by providing detailed information about the

stages that user stories go through. The proposed user story progress model distinguishes between the following states: ‘Not started’, ‘Active’, ‘Waiting for integration’, ‘Waiting for AT’ and ‘Complete’. These states can provide more accurate progress information. They reflect the effect of the technical activities on the story’s progress.

- **Process Model:** The technical activities have been re-designed in a set of process models. The aim of these process models is to provide a visual representation of how the co-ordination activities discussed in Chapter 4 can be implemented in a computer-based system.

- **Data Model:** The model represents the data necessary for developing the progress tracking system and the relationships among them.

Progress is measured through blocks of working software called stories. In order to know how far we are from completing a block (i.e. story), a story progress model is proposed that shows the stage that the story is in. Determining the story stage requires knowing the status of the source code versions which is developed to achieve the story purpose and the version model helps determine the version status.

The process model helps identify the point where a progress change takes place and then helps co-ordinate it. Finally, the data model saves the information of the various project artefacts (tasks, stories, tests, etc) and the dependencies between them.

CHAPTER 6

Evaluation

This chapter evaluates the holistic approach to developing a progress tracking system proposed in this work. In section 1.2, the hypothesis was given as:

"*Managing development progress in distributed agile projects can be supported by providing a computer-based holistic approach that co-ordinates the impact of the different technical activities on development progress, and will provide improved awareness of the actual progress to team members*."

In order to test the hypothesis, three scenarios are created. Within each scenario, a comparison is made between the old version of the scenario, where the holistic approach is not considered, and the new version of the scenario after introducing the holistic approach.

This chapter is organised as follows: section 6.1 discusses the evaluation methodology used. Section 6.2 describes the methodology used for selecting the three scenarios, while section 6.3 describes and discusses the three scenarios used for evaluation. Section 6.4 describes developing a prototype system to validate the holistic approach. Further discussion is given in section 6.5. Finally, the chapter is summarised in section 6.6.

6.1 Evaluation Methodology

6.1.1 Evaluation for Groupware Systems

The progress tracking system presented in this research is a groupware system. Ellis et al. [157] define groupware systems as:

> "*…computer-based systems that support groups of people engaged in a common task (or goal) and that provide an interface to a shared environment*."

The definition applies to the proposed progress tracking system. Therefore, its evaluation will have the same issues identified as those for evaluating groupware systems.

Evaluation of groupware has been widely considered as a difficult task and it is still an active research area in the field of computer-supported co-operative work (CSCW). Gruhn [157] observes:

> "*The almost insurmountable obstacles to meaningful, generalizable analysis and evaluation of groupware systems prevents us from learning from experience*."

A main reason why groupware is hard to evaluate is the effect of the plurality of people and their social and organizational context [158][159]. Gruhn [157] notes:

> "*Lab situations and partial prototypes cannot reliably capture complex but important social, motivational, economic and political dynamics… Field observations are complicated by the number of people involved over time at each site, the variability in group composition, and the range of environmental factors that affect the use of technology*."

Therefore, it will be difficult to use a quantitative approach to evaluate the effectiveness of a proposed groupware system (e.g. identify measurable claims such as hours saved as a result of system support for a particular activity).

A review of evaluations in 45 proposed groupware systems has been conducted by Pinelle and Gutwin [160]. The study revealed that about three quarters of groupware systems did not undergo any sort of quantitative evaluation.

To assess the value of the holistic approach in co-ordinating team members' work, it will be useful to provide qualitative-based behavioural analysis of the technical activities. This analysis is needed in order to understand how the co-ordination support enhances team members' awareness of development progress.

An analysis based on experiment may not be possible to evaluate the effectiveness of the proposed progress tracking system. The main reasons for this include the large number of dependent functionalities of the proposed progress tracking system. Implementing a complete system by one person will need considerable amount of time. In addition, allocating sufficient time to evaluate each functionality is an obstacle. This becomes impossible if there is a short time constraint for the evaluation exercise.

Another issue with experiment-based analysis is that the anticipated benefits of the proposed system may take a long time to appear [161]. The value of keeping track of the dependencies among source code artefacts, tasks, stories, and tests, may not be clear at the early stages of an agile project. It will be more obvious when the team has a large amount of data, when it is hard for the team members to understand the relationships among them.

Araujo et al. [161] observe also that it is difficult to find 'ideal' groups to conduct evaluations:

> "*It is a consensus in groupware evaluation research that groups are quite unique. Even if we try hard, it is almost impossible to find two groups with the same values to conform to our independent variables. Often we cannot find the ideal group to conduct our evaluations. To find or to build groups for evaluation is difficult and costly*."

The next sub-section introduces a scenario-based evaluation approach that is used to evaluate the holistic approach.

6.1.2 A Scenario-Based Evaluation Approach

Scenario-based evaluation (SBE) has been suggested as an effective means for the assessment of systems [162]. A key advantage of scenarios is their scalability and flexibility to account for work practices distributed over space, people, and time [163]. In addition, SBE can provide a broad understanding of the contextual interactions between users, tasks and the system features [162] [164].

The suggested evaluation approach provides an analytic comparison between the classical agile approach of performing technical activities, based on XP practices, and the proposed holistic approach. In addition, practical validation has been made by developing a prototype system for selected scenarios.

The evaluation process consists of three main parts.

- **First: Selection of scenarios**

  Real world agile projects include too many scenarios that affect the development progress. It is difficult to generate sufficient scenarios to reflect real world activity. Hence, it is more efficient and effective to generate a subset of representative scenarios that cover the main set of technical activities. In order to do this, a systematic method is required to identify suitable scenarios.

- **Second: Analysis of scenarios**

  In order to evaluate the selected scenarios, each scenario is represented twice:

  - First, with the classical XP approach. The XP approach is used as a representative of agile methods. For fair comparison, it is assumed that XP best practices are used in these scenarios. For example, in the XP project, automated AT is expected to be used for some of the tests.
  - Second, with our holistic approach.

An analytical comparison between the two scenario versions is carried out. In addition, a multi-perspective view is achieved through providing a role-oriented analysis to each scenario.

- **Third: Validation of Scenarios**

  A software system is developed to validate the holistic scenario.

6.2 Selection of Scenarios

To ensure that the selected scenarios are significant and have reasonable coverage for evaluating the holistic approach, the following methodology is adopted.

1) For each technical factor, the various events affecting progress, identified in chapter 4, are listed (Table 6-1).

2) The significance of the progress change events are evaluated on the following criteria:

- Complexity of co-ordination required.
  - Low: requires progress constraint checking only.
  - Medium: dependency is only between two team members.
  - High: dependency is among several team members.
- Frequency of progress change event.
  - Low: few times during the project.
  - Medium: several times during each iteration.
  - High: several times every day.
- Influence on development progress.
  - Low: impact on task scope only.
  - Medium: impact on one user story.

- High: impact on several user stories.

3) Scenarios are selected so that they include potential progress change events from each technical factor (i.e. source code versioning, CI and releasing, UT, and AT).

No	Technical Factor	Progress Change Event	Complexity of Co-ordination Required			Event Frequency			Influence on Development Progress		
			L	M	H	L	M	H	L	M	H
1	Source Code Versioning	Creating a new artefact belonging to un-started/incomplete tasks may change the task's state.	•			•			•		
2	Source Code Versioning	Creating a new artefact belonging to un-started/incomplete story may change the story's state.	•			•				•	
3	Source Code Versioning	Checking-out artefact version belonging to un-started/incomplete tasks may change the task's state.	•			•			•		
4	Source Code Versioning	Checking-out artefact belonging to un-started/incomplete story may change the story's state.	•			•				•	
5	Source Code Versioning	Modifying an artefact belonging to a 'complete' task may change the task's state.			•			•	•		
6	Source Code Versioning	Modifying an artefact belonging to a 'complete' story may change the story's state.			•			•			•
7	Source Code Versioning	Modifying an integrated artefact may require it to be re-integrated.			•			•	•		
8	Source Code Versioning	Deleting an integrated artefact may break the build.			•	•					•
9	CI & Releasing	An integration process has been performed that failed.			•			•			•
10	CI & Releasing	An integration process has been performed that was successful.			•			•			•
11	CI & Releasing	A set of user stories may be released while some of them have not been fully tested.			•	•					•

No	Technical Factor	Progress Change Event	Complexity of Co-ordination Required			Event Frequency			Influence on Development Progress		
			L	M	H	L	M	H	L	M	H
12	Unit Testing	Adding a unit test without re-testing it or with a 'fail' result can affect the corresponding task if it was complete.	•			•			•		
13		Modifying a unit test without re-testing it or with a 'fail' result can affect the corresponding task if it was complete.	•			•			•		
14		Deleting the only unit test for an artefact of a completed task affects the task's progress.	•			•			•		
15		A unit test may not have passed when its corresponding source code version is checked-in.	•				•		•		
16		A failed unit test prevents completing the task.	•				•		•		
17	Acceptance Testing	Adding an acceptance test without testing it, or with a 'fail' result, can affect the corresponding story if it was complete.		•		•				•	
18		Modifying an acceptance test without testing it, or with a 'fail' result, can affect the corresponding story if it was complete.		•		•				•	
19		Deleting the only acceptance test for a complete story affects the story's progress.		•		•				•	
20		Running automated acceptance testing may result in failing acceptance tests whose stories are complete.			•		•			•	
21		Running automated acceptance testing may result in passing acceptance tests whose stories are complete.			•		•			•	
22		Updating a manual acceptance test to 'fail' may cause a complete story to become incomplete.		•			•			•	
23		Updating a manual acceptance test to 'pass' may cause the story to become complete.		•			•			•	

Table 6-1. Significance of progress change events that may affect agile development progress. Key (L: Low, M: Medium, H: High).

Based on the results in Table 6-1, the following progress change events can be selected to be part of the scenarios that will be developed.

Source code versioning:

(Event 6) Modifying an artefact belonging to a 'complete' story may change the story's state.

Continuous Integration and Releasing:

Any of the following events can be selected (these events have the same significance):

(Event 9) An integration process has been performed that failed.

(Event 10) An integration process has been performed that was successful.

Unit Testing:

Any of the following events can be selected (these events have the same significance):

(Event 15) A unit test may not have passed when its corresponding source code version is checked in.

(Event 16) A complete task will be affected if one of its associated unit tests has failed.

Acceptance Testing:

Any of the following events can be selected (these events have the same significance):

(Event 20) Running automated acceptance testing may result in failing acceptance tests whose stories are complete.

(Event 21) Running automated acceptance testing may result in passing acceptance tests whose stories are complete.

It has been decided to choose the following progress events as representative of the technical factors: events 6, 10, 15 and 20. Consequently, the following scenarios are created (Table 6-2):

- To represent event 6 a '*check-in source code version*' scenario is created.
- To represent event 10, '*performing successful integration*' scenario is created.
- Because the check-in process usually includes running a unit test before checking-in the code, the '*check-in source code version*' scenario can be used to represent event 15.
- To represent event 20, '*Running Automated Acceptance testing*' scenario is created.

Scenario	Technical Factor Covered	Progress Event Covered
Scenario 1: Check-in Source Code Version	Source Code Versioning, Unit Testing	Event 6, Event 15
Scenario 2: Performing Successful Integration	Continuous Integration	Event 10
Scenario 3: Running Automated Acceptance Testing	Acceptance Testing	Event 20

Table 6-2. Scenarios used for evaluation.

These scenarios include the most significant progress change events, according to the methodology used in this section. They are also able to provide examples that show the need for each of the four key types of co-ordination activity, as identified in section 4.2.

6.3 Analysis of Scenarios

This section describes three scenarios that are independent of each other. These are: Check-in Source Code Version, Performing Successful Integration and Running Automated Acceptance Testing. Each of them has two versions: the classical XP version and the holistic approach version. The classical XP version of the scenarios are based on the best practices used for checking-in source code

[42] [165], performing integration [156] and running automated acceptance test [166].

Before describing the scenario details of each version, a general description of the scenario is given through showing pre-conditions (the state before the scenario starts), trigger (what initiates the scenario) and post-conditions (the state after completing the scenario).

6.3.1 Scenario 1: 'Check-in Source Code Version' Scenario

Pre-conditions: The source code artefact A2 has three versions: A2.1, A2.2 and A2.3. The first two versions belong to the completed stories, US1 and US2. The developer, Mike, is currently working on the third version, A2.3, as part of his work on Task T3.1 that belongs to user story US3[4] (Figure 6-1).

Trigger: Mike checks-in A2.3.

Post-conditions: The new modification made by Mike affects the user stories US1 and US2. It affects the two acceptance tests: AT1.1 that belongs to US1, and AT2.1 that belongs to US2.

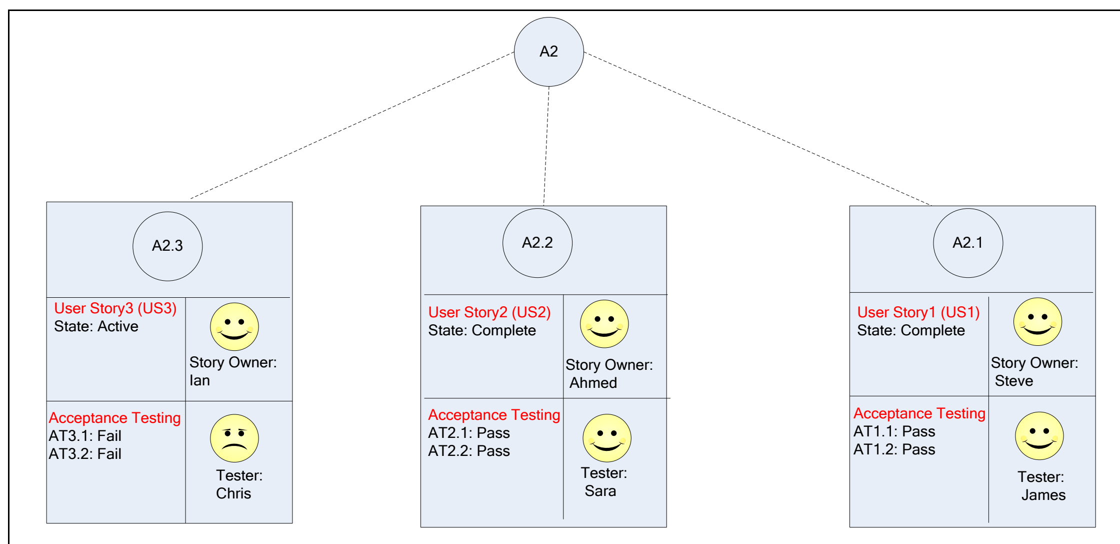


Figure 6-1. The state before the check-in process.

[4] Because of applying the practice of test-driven development (TDD) in XP, the acceptance tests AT3.1 and AT3.2 which belong to US3 are flagged as 'Fail' until team can demonstrate they have passed.

Classical XP Version of Scenario 1

1. Mike undertakes the unit testing for A2.3 and it is successful.

2. He checks in A2.3 to the versioning system.

3. He identifies US1 and US2 as a potentially affected story.

4. He looks for the affected team members (story owners: 'Steve' and 'Ahmed', testers: 'James' and 'Sara' as well as project manager) and informs them that the story might be affected.

The key scenario steps are summarised in Figure 6-2.

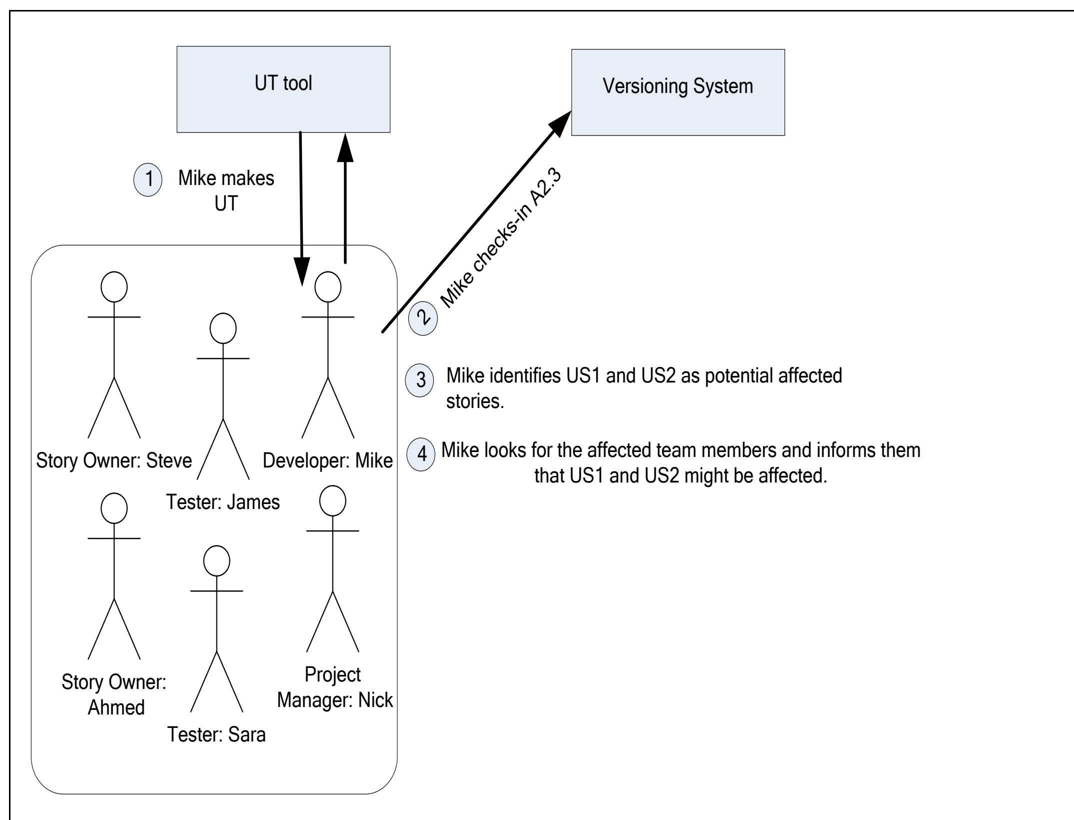


Figure 6-2. Description of the classical XP version of scenario 1.

<u>The Holistic Approach Version of Scenario 1</u>

1. Mike undertakes the unit testing for A2.3 and it is successful.

2. Mike makes a ‘check-in’ request to the tracking system.

3. The system sends a UT request to the UT tool. The test has passed; hence, A2.3 is checked-in.

4. The system retrieves stories that might be affected by introducing the new version. They are US1 and US2.

5. Mike is asked if he wants to delete any of the potentially affected stories. He does not remove any of them.

6. Progress state of US1 and US2 are changed to ‘Waiting for Integration’.

7. Notifications are sent automatically to the affected team members (story owners: ‘Steve’ and ‘Ahmed’, testers: ‘James’ and ‘Sara’ as well as project manager).

The key scenario steps are summarised in Figure 6-3.

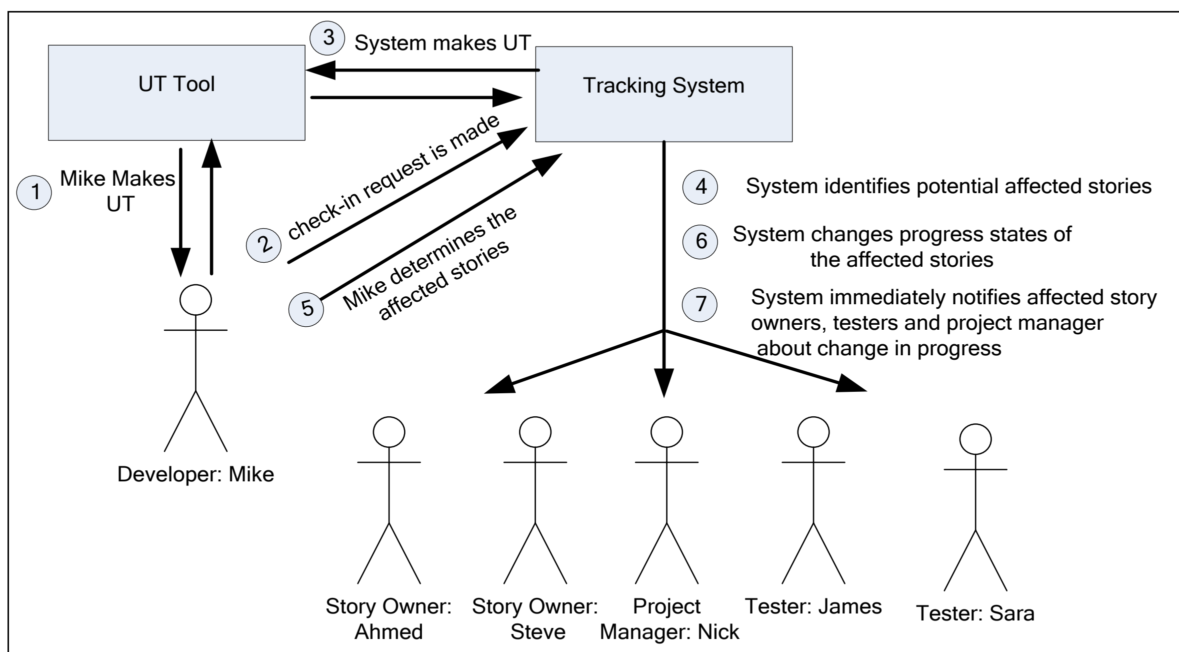


Figure 6-3. Description of the holistic approach version of scenario 1.

Scenario 1 Analysis

As mentioned earlier, the check-in scenario covers the progress change events 6 and 15. These events require performing all the four types of co-ordination activities identified in section 4.2:

- **Checking progress constraints:** ensure that all corresponding unit tests are successful.
- **Identifying potential sources of progress change:** when the shared artefact is updated, the stories that have been affected must be identified.
- **Reflecting progress change in the tracking system:** the progress state of the affected stories has to be changed.
- **Finding and notifying team members affected by a potential progress change:** the affected team members (e.g. story owner and tester responsible for the acceptance testing for the story) must be found and notified.

Table 6-3 compares the holistic approach with the classical XP approach, based on the four co-ordination activities needed to manage progress change events 6 and 15.

The comparison shows that the holistic approach provides better awareness of the actual work completed by the developers' tasks. It immediately identifies the potential change in progress resulting from the check-in process. In addition, affected team members are immediately informed about the change. Therefore, the holistic approach can help team members become aware earlier of the sources of the potential defects that may cause a project delay.

Co-ordination Activity	The Classical XP Approach	The Holistic Approach
Checking progress constraints	• It is up to the developer to make the UT. • He may forget to run the UTs or may not follow the practices. This may lead to shared code that contains errors.	• Automatic verification is carried out. • Automating the process ensures that only unit-tested code is shared among the developers.
Identifying potential sources of progress change	• Affected stories may be identified but it can be difficult for the developer working in the distributed project to identify which stories. It is also time consuming. • If affected stories are not identified earlier by the developer, then these stories may be unidentified until the next AT time. • If the AT is automated, the team members would still need to investigate the source of the problem. In addition, not all ATs can be automated. • A manual AT may allow for a long defect life before it is discovered. This may cause the introduction of new defects to the project.	• Potentially affected stories are automatically identified once the developer checks-in the source code. • The holistic approach can provide better visibility of the actual progress. It immediately identifies the potential change in progress resulting from the check-in process. By doing so, the holistic approach will help the team members identify the potential source of the defects that may cause a project delay rather than waiting until they are discovered during AT time.
Reflecting progress change in the tracking system	• Team members usually share the new progress state informally, not in the tracking system.	• The change of state is reflected in the tracking system. • It increases the entire team's awareness about the project state.
Find and notify the affected team members	• If the developer identifies an affected story, he will need to determine who must be contacted and then will need to share the information with them informally (e.g. by e-mail or during the next video-conferencing meeting). • If the developer is unable to identify some of the affected stories, this activity will not be carried out until a defect is discovered in the AT.	• The affected story owners and testers are notified automatically by the system once the versioning activity is used. In addition, the project manager is immediately informed about the change in progress.

Table 6-3. Analysis of the co-ordination support in scenario 1.

The scenario involves participation of developers, testers, story owners and the project manager. The benefits that each individual may achieve from introducing the holistic approach is assessed through a role-oriented analysis (Table 6-4).

Role of the Team Member	The Classical XP Approach	The Holistic Approach
Developer	● checks-in an artefact version. ● has to understand how his modification affects other team members. ● has to find and notify affected team members.	● checks-in an artefact version ● determines the potentially affected stories suggested by the tracking system.
Tester	● will not know the effect until AT is made or contacted by the developer. ● If an acceptance test failed, he has to find and notify story owner about change in progress ● has to trace changes to detect source of the failure.	● is informed immediately about potential source of defect. He does not need to trace changes to detect source of the defect.
Story Owner	● will not know the effect until AT is made or contacted by the developer.	● is informed immediately about the progress change in his story.
Project Manager	● will not know the effect until AT is made or contacted by the developer.	● is informed immediately about the progress change.

Table 6-4. Role-Based Analysis of scenario 1.

6.3.2 Scenario 2: Performing Successful Integration

Pre-conditions: Ten new versions have been developed since last integration. The functionalities required for user story US1 have been implemented and the story is waiting for its new versions, A5.1 and A6.1, to be integrated. In addition, the functionalities required for user story US2 have been implemented and the story is waiting for its new versions, A8.1 and A9.1, to be integrated. The user stories US3 and US4 are still active (Figure 6-4).

Trigger: Additionally to the nightly build practice that the team follows, Sally would like to initiate an integration process to ensure that source code does not include any integration problems at the moment.

Post-conditions: The integration is successful and the user stories US1 and US2 become ready for acceptance testing.

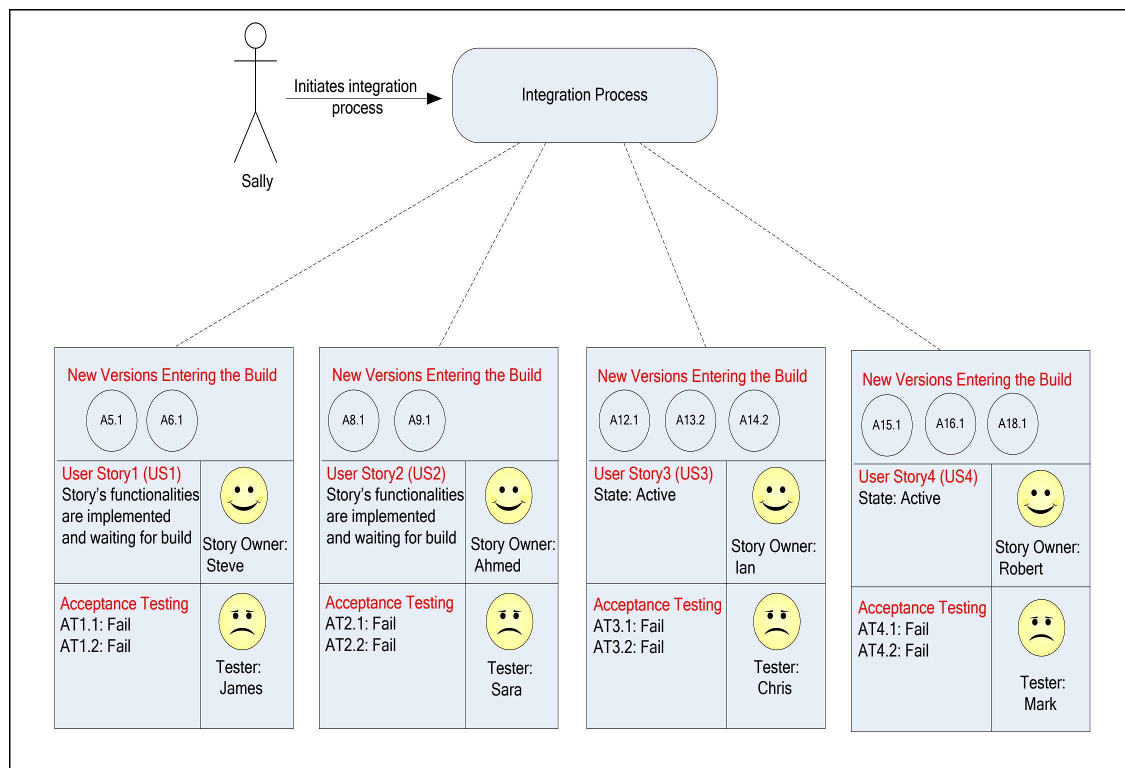


Figure 6-4. The state before performing the integration.

<u>Classical XP Version of Scenario 2</u>

1. Sally clicks on ‘Perform Integration’ in the continuous integration (CI) system.

2. The integration system retrieves the new artefact versions from the versioning system and performs the integration.

3. The integration system returns the result to Sally and sends generic notifications of the integration result to team members.

4. Team members need to figure out which story functionalities are completely implemented and integrated in the current build and then need to be acceptance-tested.

The key scenario steps are summarised in the following diagram (Figure 6-5):

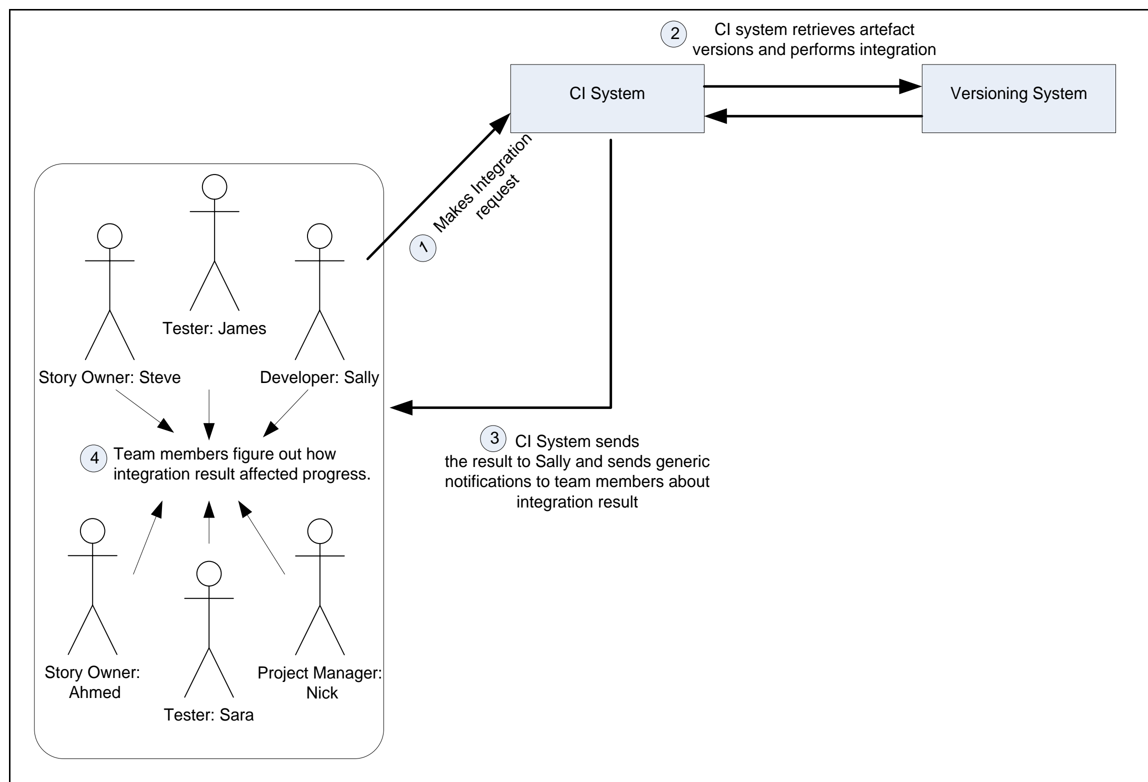


Figure 6-5. Description of the classical XP version of scenario 2.

<u>The Holistic Approach Version of Scenario 2</u>

1. Sally clicks on 'Perform Integration' in the tracking system.

2. System retrieves the last UTVs of the recently updated artefacts and the last IVs of the non-recently updated artefacts and sends an integration request to the continuous integration (CI) system.

3. System receives 'Successful' result from the CI system and updates the UTV versions to 'IV'.

4. System checks if there are any 'Waiting for Integration' user stories. It moves the stories US1 and US2 to 'Waiting for AT'.

5. Generic notifications are sent to all team members to raise awareness of the integration result. In addition, specialised notifications, clarifying the new state of US1 and US2, are sent to those responsible for US1 and US2, story owners (Steve and Ahmed) and testers (James and Sara) as well as the project manager.

The key scenario steps are summarised in the following diagram (Figure 6-6):

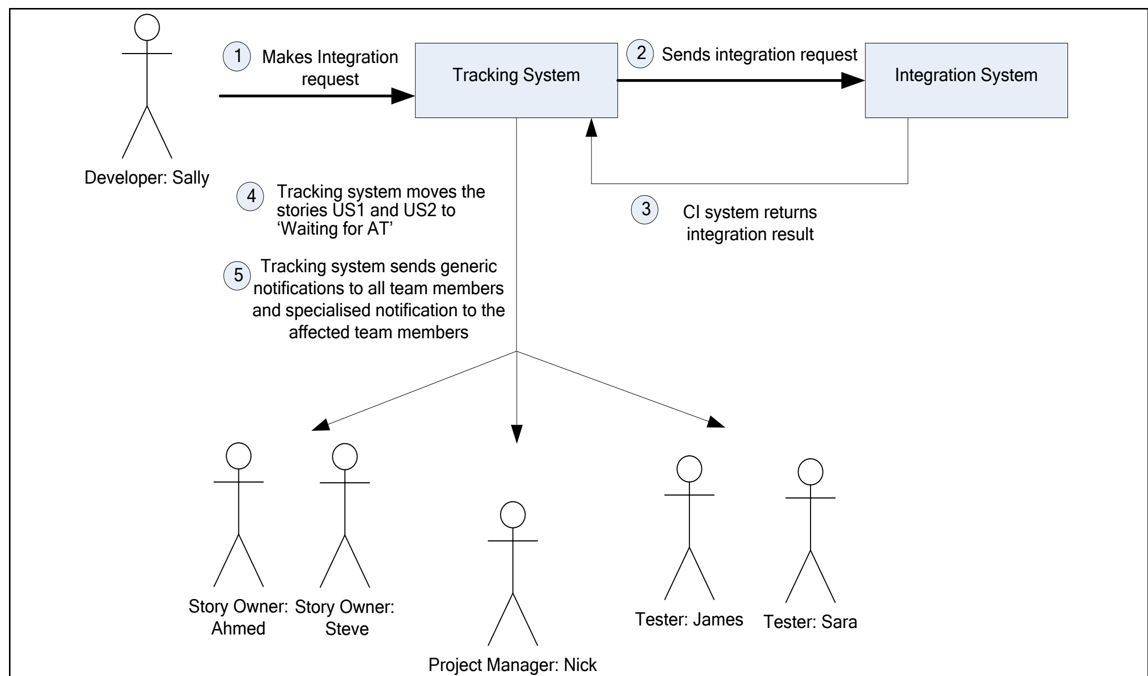


Figure 6-6. Description of the holistic approach version of scenario 2.

Scenario 2 Analysis

The 'Perform Successful Integration' scenario covers change event 10. This event requires performing three of the co-ordination activities identified in section 4.2:

- **Identifying potential sources of progress change:** an integration 'pass' result should contribute to making progress on the stories that are completely implemented and have associated versions entering the build.
- **Reflecting progress change in the tracking system:** the progress state of the affected stories has to be changed.
- **Finding and notifying team members affected by a potential progress change:** story owner and tester have to be located and notified that the story is now ready for acceptance testing.

Table 6-5 compares the holistic approach with the classical XP approach, based on the three co-ordination activities needed to manage progress change event 10. The holistic approach provides better visibility of the actual work completed by team members. It automatically identifies the affected stories and hence team members will not need to spend time recognising how the integration result affects their work progress. When integration passes, relevant story owners and testers become aware immediately that their stories have become ready for acceptance testing. The automatic notification helps in making the acceptance test as early as possible in the development cycle.

Co-ordination Activity	The Classical XP Approach	The Holistic Approach
Identifying potential sources of progress change	● Team members need to figure out how the integration result affects progress. ● It can be difficult for team members working in the distributed project to realise which stories have been affected.	● Potentially affected stories are automatically identified once the integration is performed. • Provides better visibility of the actual progress. Immediately identifies the potential change in progress resulting from the integration process.
Reflecting progress change in the tracking system	• Team members usually share the new progress state informally, not in the tracking system.	● The integration effect is automatically reflected in the tracking system. • It increases the entire team's awareness about the project state.
Finding and notifying team members affected by potential progress change	● It is done in an ad-hoc manner. ● The affected story owners and testers may be in different sites. This may make it difficult to identify who should be notified. ● A delay in making the acceptance testing may take place because affected team members do not know that the story is ready for acceptance testing.	● The affected story owners and testers are notified automatically by the system once the integration process is completed. ● The automatic notification raises awareness that the stories are available for acceptance testing, thus increasing the opportunity to run the acceptance tests earlier in the development cycle.

Table 6-5. Analysis of the co-ordination support in scenario 2.

Similar to Scenario 1, this scenario involves participation from a developer, story owners, testers and project manager. A role-oriented analysis is provided in Table 6-6.

Role of the Team Member	The Classical XP Approach	The Holistic Approach
Developer	● performs integration	● performs integration
Tester	● needs to figure out which story functionalities are completely programmed and integrated in the current build.	● is informed immediately about which stories have become ready for AT.
Story Owner	● may not recognise how the integration result affects his story progress.	● is informed immediately about change in his story progress.
Project Manager	● will know the integration result but will not know how the result affects the development progress.	● The approach allows him to realise the effect of the integration on the development progress.

Table 6-6. A Role-based analysis of scenario 2.

6.3.3 Scenario 3: Running Automated Acceptance Testing

Pre-conditions: The acceptance tests AT1.1, AT2.1 and AT3.1 are passed. These tests belong to the completed stories US1, US2 and US3 respectively (Figure 6-7).

Trigger: As part of a regression testing, the test leader, Sam, runs automated acceptance testing with the three acceptance tests (AT1.1, AT2.1, AT3.1).

Post-conditions: The acceptance tests AT1.1 and AT2.1 failed due to recent modifications to shared code belonging to US1 and US2.

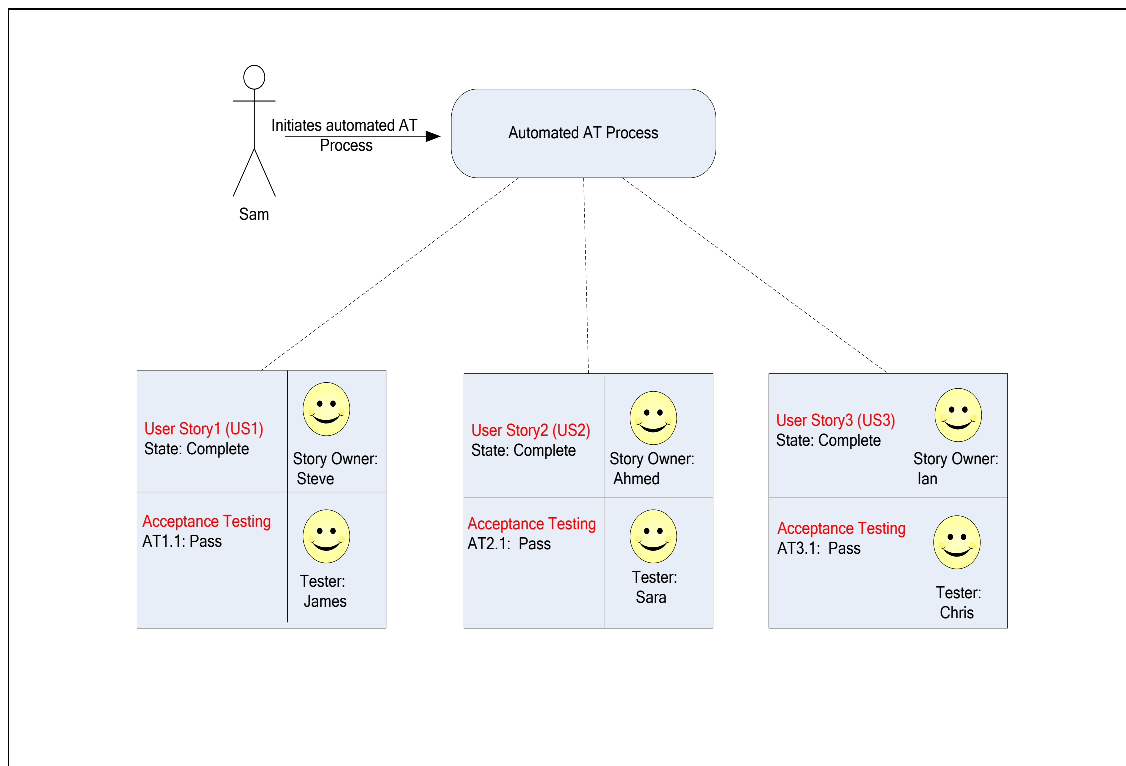


Figure 6-7. The state before the testing process.

<u>Classical XP version of Scenario 3</u>

1. Sam initiates automated acceptance testing using the AT tool.
2. The AT tool performs the tests and returns the results to Sam.
3. Sam finds and then notifies the affected team members.

The key scenario steps are summarised in the following diagram (Figure 6-8):

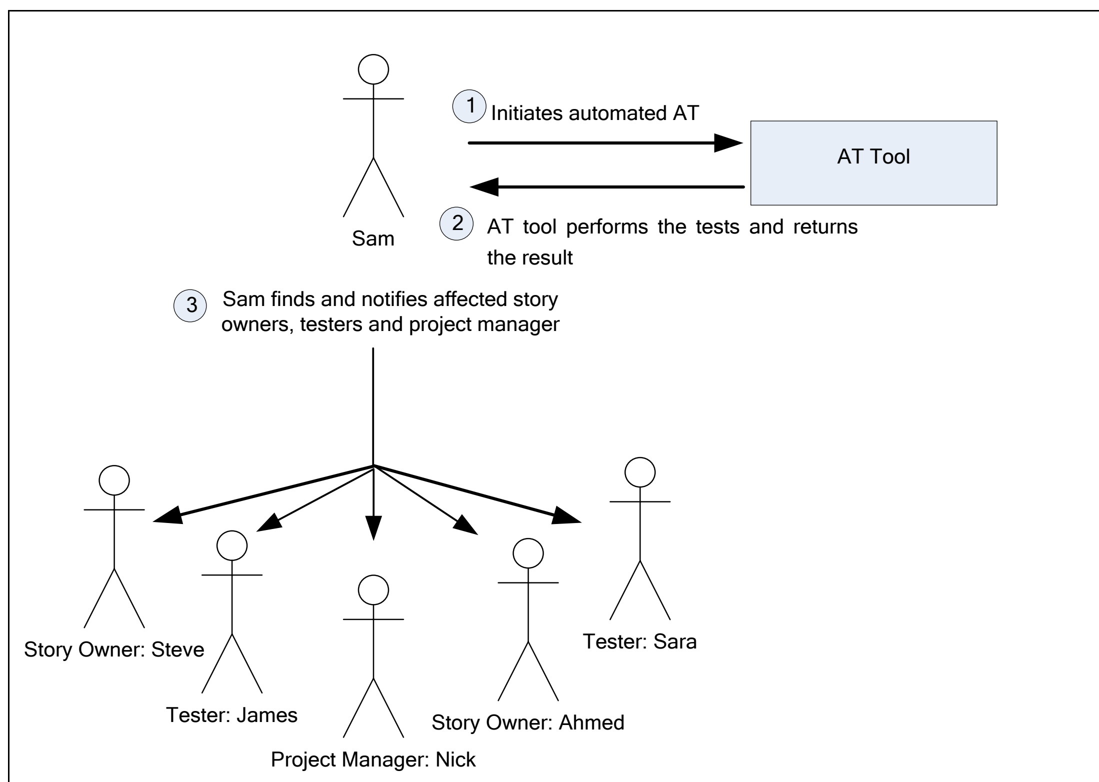


Figure 6-8. Description of the classical XP version of scenario 3.

The Holistic Approach Version of Scenario 3

1. Sam initiates automated acceptance testing using the tracking system.
2. The tracking system sends request to the AT tool and then receives the test result.
3. As the result shows that the acceptance tests AT1 and AT2 failed, the tracking system changes the state of the user stories US1 and US2 to 'Waiting for AT'.
4. The tracking system provides the result to Sam and automatically sends notifications to the affected team members (story owners: Steve and Ahmed, testers: James and Sara, and the project manager, Nick).

The key scenario steps are summarised in the following diagram (Figure 6-9):

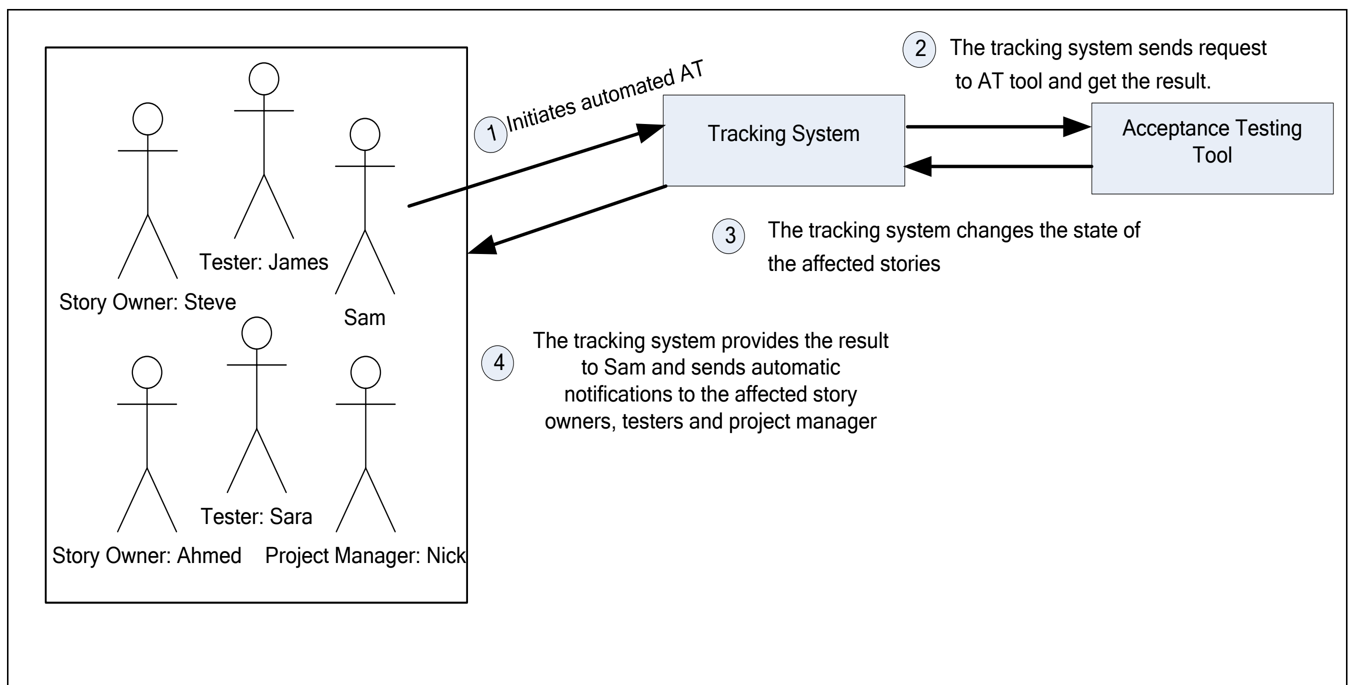


Figure 6-9. Description of the holistic approach version of scenario 3.

Scenario 3 Analysis:

The 'Running Automated AT' scenario covers change event 20. This event requires performing three of the co-ordination activities identified in section 4.2:

- **Identifying potential sources of progress change:** if running automated acceptance testing has led to failed acceptance tests and if these acceptance tests belong to completed stories, these stories become incomplete.
- **Reflecting progress change in the tracking system:** the progress state of the affected stories has to be changed.
- **Finding and notifying team members affected by a potential progress change:** story owners and testers have to be located and notified that the affected stories have failed acceptance tests.

Table 6-7 compares the holistic approach with the classical XP approach, based on the three co-ordination activities needed to manage change event 20. This scenario shows that the holistic approach allows the tracking system to identify the influence of failed acceptance tests on development progress. If an acceptance test fails, the system automatically changes the state of the corresponding user story and notifies the story users.

By replacing the manual method, the holistic approach can provide better awareness to team members about the real progress of development. The impact of failed acceptance tests is formally reflected in the tracking system. In addition, the affected story owners and testers, as well as project manager, are automatically found and notified.

Co-ordination Activity	The Classical XP Approach	The Holistic Approach
Identifying potential sources of progress change	● The person who made the automated AT may not know which stories have been affected. This is likely if the failed acceptance tests belong to stories created at different sites.	● Identified automatically by the tracking system.
Reflecting progress change in the tracking system	● Progress change is not reflected in the tracking system.	● The effect of the automated AT is automatically reflected in the tracking system. ● It increases the entire team's awareness of the project state.
Finding and notifying team members affected by potential progress change	● It is done in an ad-hoc manner. ● The affected story owners and testers may be at different sites. This may make it difficult to identify who should be notified. ● A delay in resolving the defects may take place because affected team members may not be notified about the failed tests.	● The affected story owners and testers are notified automatically by the tracking system once the AT process is completed. ● The automatic notification helps in resolving the defects early in the development cycle.

Table 6-7. Analysis of the co-ordination support in scenario 3.

This scenario involves participation of the test leader, testers, story owners and project manager. An evaluation of the benefits that each of them can achieve is assessed in the role-oriented analysis presented in Table 6-8.

Role of the Team Member	The Classical XP Approach	The Holistic Approach
Test Leader	● performs the automated AT. ● has to find and notify the affected team members.	● performs the automated AT.
Tester	● will not be able to investigate the source of the problem until he is notified by the test leader.	● is informed immediately about failed test.
Story Owner	● will not be able to investigate the source of the problem until he is notified by the test leader.	● is informed immediately about change in his story progress.
Project Manager	● will not know the actual progress until he is notified by the test leader.	● will be notified automatically about the progress change.

Table 6-8. A Role-based analysis of scenario 3.

6.4 Validation of the Holistic Approach

A research prototype system has been developed to demonstrate the feasibility of the proposed holistic approach to developing a progress tracking system. It validates the holistic approach version of the three scenarios provided in the previous section and ensures that a computer-based system is capable of demonstrating them.

The NetBeans IDE [167] has been used to develop the application, while MySQL [168] is used as the backend database. Both are popular tools used for creating computer applications.

6.4.1 System Database

The data model created in Chapter 5 has been translated into a progress tracking system database (Figure 6-10). This database is sufficient to keep track of the basic data needed to demonstrate the value of the proposed holistic approach.

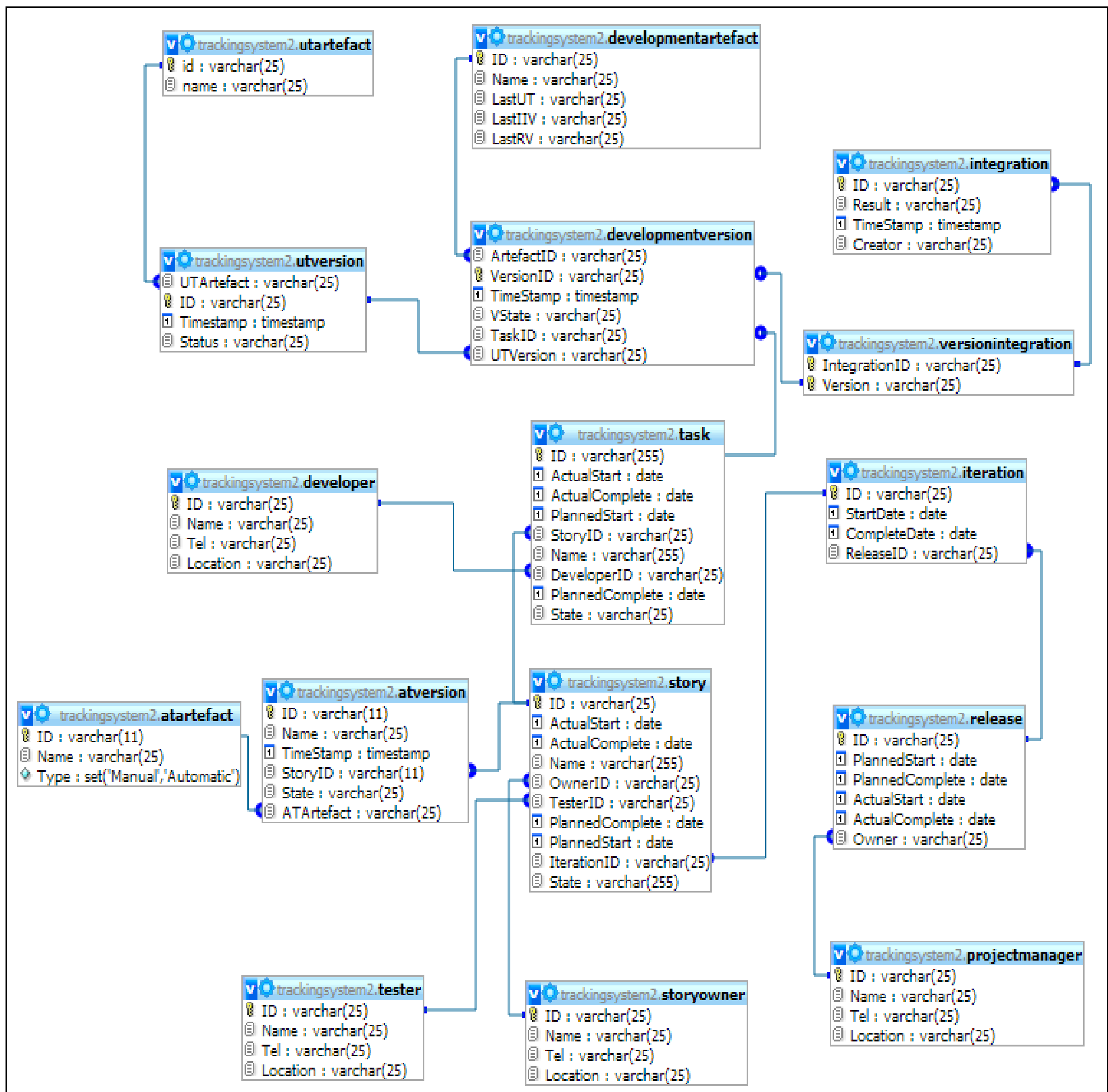


Figure 6-10. Progress Tracking System Database.

6.4.2 Implementation of Scenario 1

The prototype implementation does not provide complete functionalities for the tracking system but demonstrates only that a computer system is able to perform the steps involved in each of the three holistic approach scenarios described in this chapter. The implementation for each scenario consists of a sequence of screens, where each screen represents one step of the scenario described earlier. Each screen displays the output that results from moving from one step to another. A timer is used to move the screens forward.

Here, we explain in detail the various steps involved in the holistic approach version of scenario 1 (Check-in Source Code Version). The implementation description of the other two scenarios is described in Appendix C.

Scenario 1 starts with the following initial data set:

- The source code artefact A2 has three versions: A2.1, A2.2 and A2.3.
- A2.1 and A2.2 belong to the completed stories, US1 and US2.
- The developer, Mike, is working on the third version, A2.3, as part of his work on Task T3.1, which belongs to user story US3.

The implementation of the six steps involved in the holistic version of the check-in scenario is discussed below.

1- Mike makes a ‘check-in’ request to the tracking system (Figure 6-11):

The versions that the developer updates as part of his work on Task 3.1 can be achieved through the following query:

```
SELECT a.versionID
FROM Developmentversion a
WHERE a.taskID='T3.1
```

Scenario 1 (Check-in Source Code Version)
File Help
Task Id: T3.1
Name: Design input form
State: Active
Save
Artefacts Currently Working-on
A2.3 Edit Check-in
Developer Checks-in A2.3
Timer
Next 1 Reset

Figure 6-11. Implementation of scenario 1, step 1.

2- After the system checks the corresponding unit test passed, A2.3 is checked-in (Figure 6-12).

Scenario 1 (Check-in Source Code Version)
File Help
System checks that the version is unit-tested
The following information are retrieved from the database

Version ID	UTVersion	VState	Task ID
A2.3	ut3	TV	T3.1

Unit Test ID	Status
ut3	Pass

The system checks the unit test passed
Timer
Next 2-a Reset

Figure 6-12. Implementation of scenario 1, step 2.

The system retrieves the state of the corresponding unit test through the following query:

```
SELECT u.id, u.status
FROM Utversion u , Developmentversion a
WHERE a.versionID='A2.3' and a.uTVersion=u.id
```

If the unit test is in the 'pass' state, the artefact version can be checked-in. The check-in process promotes the version to 'UTV' state.

```
UPDATE Developmentversion a
SET a.vState='UTV'
WHERE a.versionID='A2.3'
```

3,4. The system retrieves stories that might be affected by introducing the new version. They are US1 and US2. Mike is asked if he wants to delete any of the potentially affected stories from the list. He does not remove any of them.

In order to obtain this information, the system has to check stories that use the same artefact and are now in 'Waiting for AT' state or in 'Complete' state:

```
SELECT story.id, story.name
FROM task, story
WHERE (story.state = 'Complete' or story.state =
       'Waiting for AT')and task.storyID=
       story.ID and task.id in
          (SELECT tasked
           FROM Developmentversion
           WHERE artefactid =
                (SELECT artefactid
                 FROM  Developmentversion
                 WHERE versionid= 'A2.3'))
```

The query result retrieves the stories US1 and US2 as they both are complete and updated the artefact A2 (Figure 6-13).

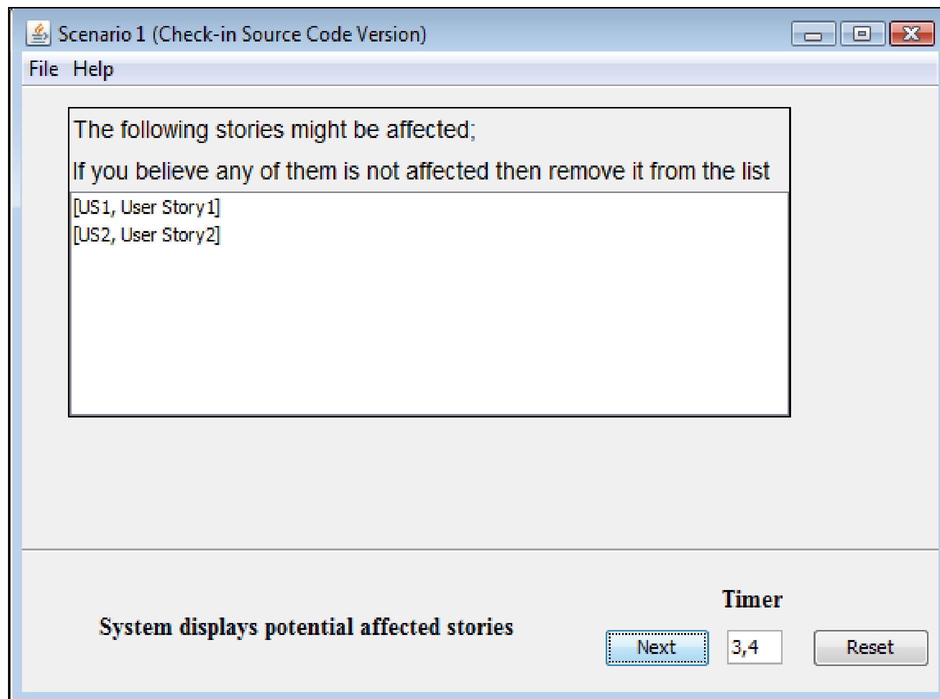


Figure 6-13. Implementation of scenario 1, steps 3,4.

5. As stated in the scenario description, Mike believes both stories US1 and US2 are affected; hence, the system updates their states to 'Waiting for Integration' (Figure 6-14) using the following SQL update statement.

```
UPDATE Story s
SET s.state='Waiting for Integration'
WHERE ((s.id='US1')or (s.id='US2'))
```

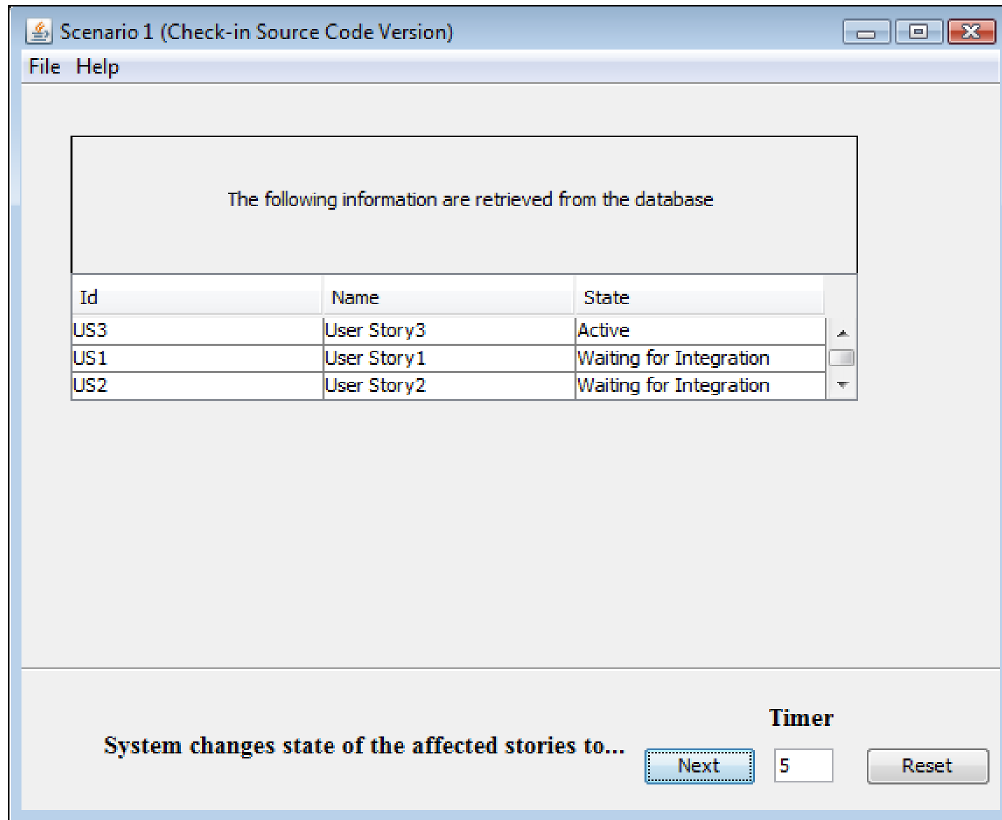


Figure 6-14. Implementation of scenario 1, step 5.

6. Notifications are sent automatically to the affected team members (story owners: 'Steve' and 'Ahmed', testers: 'James' and 'Sara' as well as the project manager).

In order to retrieve the affected story owners, the following query is created:

```
SELECT ss.id StoryID, so.name StoryOwner,
       so.location OwnerLocation
FROM story ss, storyowner so
WHERE ss.ownerid=so.id and  (ss.id = 'US1' or
      ss.id = 'US2')
```

and in order to retrieve the affected testers, the following query is created:

```
SELECT ss.id StoryID, t.name Tester,
       t.location TesterLocation
FROM   story ss, tester t
WHERE  ss.testerid=t.id and  (ss.id = 'US1'
       or ss.id = 'US2')
```

The query result shows that the story owners (Steve and Ahmed) and testers (James and Sara) are affected.

A notification message is sent automatically to each of the affected team members. Example of such message is shown in Figure 6-15. It shows the affected story, which caused the change in progress and in which site Mike works in.

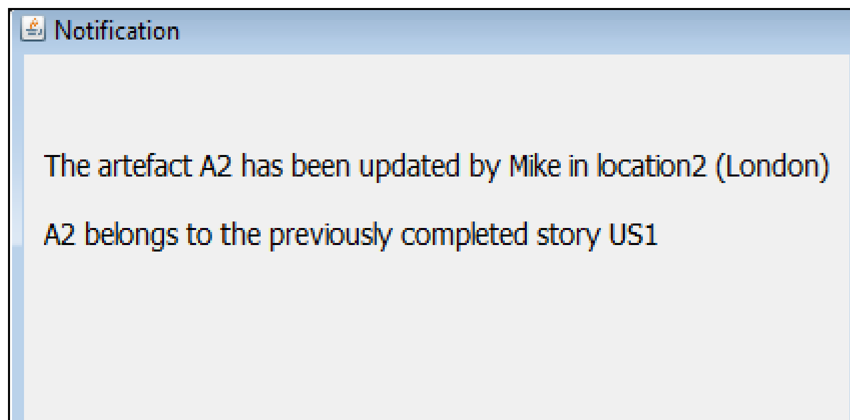


Figure 6-15. Implementation of scenario 1, step 6.

6.4.3 Validation Discussion

Implementation of the three scenarios has shown several results. These results are discussed below.

- The capability of keeping track of the dependencies between project artefacts (tasks, stories, tests, code etc):

  Each project artefact is represented in a table in the database and a logical representation of the relations between the tables is made. Once one of the artefacts changes, it becomes possible to know which of the other artefacts have been affected.

- The capability of storing changes to the source code versions and moving them from one state to another based on versioning activities such as check-in process:

  Implementation of the check-in scenario involved moving the TV version that Mike was working on in his private workspace to UTV. Such change to the version state allows sharing the version with the other developers, where they can use the version in confidence that it has been unit-tested.

- The capability of checking progress constraints:

  Before checking-in the source code version A2.3, the system carries out an activity that makes sure that the version is unit-tested. This guarantees that only the unit-tested code is shared amongst team members.

- The capability of reflecting the impact of technical activities carried out by team members on development progress state:

  In implementation of the check-in scenario, the system discovered what stories have been potentially affected due to the recent change introduced by Mike. This has been discovered through querying the database about the stories that use the same artefact and are in ‘Waiting for AT’ state or in ‘Complete’ state. As the developer believes that all the suggested stories are affected, an update statement is made to change the stories’ states to ‘Waiting for Integration’.

- The capability of identifying affected team members and targeting the co-ordination to those who are impacted:

  After determining the affected stories, it becomes easy to identify the affected people. This is because the information on the story owner and tester for each user story is registered in the database.

6.5 Further Discussion on Evaluation

6.5.1 Overcoming the Limitations of the Informal Methods

The three scenarios of the classical extreme programming approach show that the impact of the technical activities on development progress is completely managed manually in an ad-hoc manner. It is the team members’ responsibility to identify and co-ordinate progress change events.

Section 3.3 highlighted the main limitation of informal methods used to manage development progress of distributed agile projects as the impact of technical

activities on progress may not be fully recognised by the team members. As an agile project grows, the number of dependencies between progress information and technical objects (e.g. source code artefacts and tests) becomes uncontrollable and difficult to keep track of. This is true for the three scenarios introduced in this chapter, although these scenarios consider only a few dependencies.

The computer-based holistic approach overcomes the limitations of the informal methods by providing a system architecture that allows the impact of technical activities on development progress to be captured. The automatic identification and co-ordination of progress change events compensate for human deficiencies.

The holistic approach does not replace the need for informal communication among team members as it also supports raising the awareness of the changes that may affect development progress. However, depending completely on humans to capture and co-ordinate the different types of change is unrealistic. One major cause is the complexity involved in understanding the impact of changes on development progress.

6.5.2 Overcoming the Limitations of the Formal Methods

Although there are computer systems which include many mechanisms to support managing progress of distributed agile projects, as discussed in section 3.2, all these systems fail in identifying the impact of technical activities on development progress. This is because they rely on changes in progress, caused by the technical activities, to be flagged in the system by team members.

The proposed holistic approach extends the scope of current progress tracking systems. It makes the tracking system identify progress change events, not only through user inputs, but also through automatic identification. It then helps raise team members' awareness of the progress state by providing the necessary co-ordination for each progress change event.

6.6 Summary

This chapter has evaluated the holistic approach to developing a progress tracking system for distributed agile teams. A scenario-based approach has been used as an evaluation methodology. The following three scenarios were used:

- Check-in Source Code Version
- Performing Successful Integration
- Running automated Acceptance Testing

The comparisons between the execution of these scenarios when the classical XP approach is used, and when the holistic approach is introduced, revealed that better awareness of progress can be achieved with the holistic approach. The strength of the holistic approach is that it provides automatic identification and co-ordination of progress change events.

A proof-of-concept prototype was developed to ensure that a computer system is capable of demonstrating the holistic approach. The implementation demonstrated that the holistic approach scenario can be made. The database schema and the SQL queries were able to identify the change in progress and to provide the necessary co-ordination.

CHAPTER 7

Conclusions

This chapter concludes the research reported in this thesis. It will discuss the achievement of the research against its hypothesis and objectives (section 7.1), future directions for further improvement (section 7.2), and will finally discuss the overall contribution of the research (section 7.3).

7.1 Achievement of the Research Objectives

Section 1.2 included a statement of the hypothesis of the research as:

"*Managing development progress in distributed agile projects can be supported by providing a computer-based holistic approach that co-ordinates the impact of the different technical activities on development progress, and will provide improved awareness of the actual progress to team members*."

The research conducted in this thesis and the approach documented in the previous chapters tested this hypothesis to the point where it is possible to say that it does indeed hold true. The holistic approach helps distributed agile teams identifying potential sources of progress change and helps co-ordinate it with other team members. This overcomes the limitation of the informal methods, where team members completely rely on their understanding of how carrying out technical activities may change the progress. The holistic approach also

overcomes the limitations of the formal methods. These methods use computer systems merely for registering progress information but do not help team members in identifying the point where a progress change event occurs and obviously do not provide the co-ordination support necessary to deal with such events.

Providing an effective approach that incorporates the impact of the technical activities on development progress improves the awareness of distributed agile teams regarding the actual progress. Team members will no longer struggle in understanding their change impact on development progress alone, but will be provided with a system that help them achieve that. In addition, they will not rely on static information about progress registered in a progress tracking system, but will be updated continuously with relevant information about progress changes occurring to their work. Notifications regarding the changes in the progress are targeted to those affected team members, which can help solving the problem of information overload (i.e. having too much information where it becomes difficult to understand how they relate to a team member's work).

Section 1.2 also identified the objectives that needed to be satisfied to achieve the research aim of developing a computer-based holistic approach to managing progress in a distributed agile development. These objectives are reviewed below.

Objective 1: *Defining the concept of progress in the agile approach and the difference in progress tracking between the agile approach and the traditional (plan-driven) approach. This includes identifying the key technical factors affecting agile development progress.*

Section 2.2.2 explained the meaning of progress in the agile approach and how it is different from the plan-driven methodologies. While the development progress in plan-driven methodologies is based on the completion of deliverables such as the requirement specification document and analysis and design diagrams, the

agile approach considers the amount of 'working software' as the primary measure of progress.

Section 2.3.1 provided a proposition of the need for effective approach to incorporate the impact of the technical factors (UT, AT, CI and Releasing, and source code versioning) on development progress. This is because these factors impact the progress towards working software. The identification of these factors has been derived from analysing the factors that contribute to producing working software. Each of these factors comprises a set of technical activities affecting agile development progress (see Table 7-1).

Unit Testing (UT)	Acceptance Testing (AT)	Continuous Integration (CI) & Releasing	Source Code Versioning
• Create a new UT	• Create a new AT	• Perform integration	• Create an artefact
• Update existing UT	• Update existing AT	• Make a release	• Modify an artefact
• Delete UT	• Delete AT		• Delete an artefact
• Run UT	• Run AT		

Table 7-1. Key technical activities affecting agile development progress.

The popularity of the technical factors among the agile methods was surveyed in section 2.3.2. The results showed that most agile methods recommended using them.

The discussion in sections 2.2.2, 2.3.1, and 2.3.2, contributed to explicitly defining an agile philosophy of development progress and the factors affecting it, which has been the basis for developing a progress tracking system for agile teams.

Objective 2: *Surveying how well the informal methods and the formal methods manage progress in a distributed agile development.*

Chapter 3 discussed the current approaches used to manage development progress in distributed agile environments. It discussed two approaches: informal methods and formal methods.

Informal methods rely on humans to identify and co-ordinate development progress in an ad-hoc manner. The main informal methods discussed were synchronous communication, asynchronous communication, daily tracker, information radiators and cross-location visits.

The limitation of the informal methods is that the impact of the change may not be fully recognised by the team members. This is because of the difficulty of understanding how the work of one team member at one site influences the work of another team member at a different site.

Formal methods use automatic mechanisms for storing, retrieving and manipulating progress information. The formal methods include Wikis and spreadsheets, traditional project management tools, and agile project management (APM) tools. These methods were reviewed but with particular focus on APM tools. A review of 30 APM tools has been carried out by using several methods (i.e. working on trial versions, watching demos, reading the formal description of the tools and asking questions through community boards). The use of a variety of methods was useful as some information that could not be found by one method is gathered by others. The review revealed a number of mechanisms available to assist in supporting the management of distributed agile development. The key progress tracking mechanisms in these tools are: web-based task board, progress reporting, time tracking, acceptance testing (AT) tracking and progress notifications.

The limitation of the formal approach is that the computer systems used are static and completely rely on team members to report changes in progress.

The analysis of the informal and formal methods achieved the second objective and revealed that they are insufficient to manage distributed agile development progress. As a result, a computer-based holistic approach was suggested. The progress tracking system has a holistic view as it realises the effects of changes from the users (team members) and also from the various technical systems that cause progress change.

Objective 3: *Identifying the co-ordination support required for managing development progress. This includes analysing the various events that cause change in progress.*

Technical activities may cause progress change events that require performing further co-ordination activities. Chapter 4 identified 23 events that may cause change in progress. The progress events identified are comprehensive enough to cover all the technical activities carried out by team members (UT activities, AT activities, CI and releasing activities, and source code versioning activities).

With each progress event, the co-ordination support necessary to manage the event has been explicitly identified. The identification of co-ordination requirements is based on the four key types of co-ordination activity required for managing progress in agile projects: checking progress constraints, identifying potential sources of progress change, reflecting progress change in the tracking system, and finding and notifying team members affected by potential progress change.

Objective 4: *Designing a computer-based system capable of providing the necessary co-ordination. Computer-based mechanisms have to take into consideration the impact of the technical activities on progress.*

Chapter 5 proposed an approach for designing a progress tracking system for distributed agile teams. The system keeps track of the impact of the technical activities by placing them under control of the tracking system. This has been achieved by integrating the versioning functionality into the progress tracking system and linking the UT tool, AT tool and CI tool with the progress tracking system.

A version model was created to show the progress of each source code version. Because development progress in agile approach is based on the state of the source code, the model helps by informing users of the actual state of a task/story. The progress states of tasks/stories are now based on their corresponding source code. With this novel design, it is no longer possible to claim that a task is complete while its corresponding source code artefacts have not been unit tested. Moreover, it is no longer possible to claim that a user story is complete while its corresponding source code artefacts have not been integrated, or the user story as a whole has not been accepted by the customer. The version model helps to change the thinking about progress tracking and moves it from being traditional, where it is merely based on completion of duties, to becoming more agile, where it is based on the maturity of source code and how far it is from being 'working software'.

The user story progress model provided detailed information about the stages that user stories go through. User stories may assume one of the following states: 'Not started', 'Active', 'Waiting for integration', 'Waiting for acceptance testing', and 'Complete'. Unlike the traditional story progress states, which assume that a story is in 'Complete' state once its functionalities are implemented, the proposed model makes explicit differentiation between three different states: a story's functionalities may only be implemented ('Waiting for Integration' state), a story's functionalities may be implemented and also

integrated ('Waiting for acceptance testing' state), and a story's functionalities may be implemented, integrated and accepted by the customer ('Complete' state). The value of this differentiation is that the progress measure now becomes based on an agile perspective of what is considered a complete story. This supports providing a more realistic view of the actual state of the software project. Another value of the proposed user story progress model is that it reflects the impact of the technical activities on development progress. For instance, modifying shared source code belonging to a completed story may result in the story being incomplete. The modification effect on the story progress has to be explicitly shown by the progress tracking system. The team needs to know that this story has become incomplete and thus may need to be re-integrated and acceptance-tested again.

The design of a progress tracking system required the provision of process models to cover the various technical activities. The modelling of these processes is considered an integral part of designing the progress tracking system, as it provides a visual representation of how the co-ordination activities necessary for each technical activity can be implemented in a computer-based system. This includes providing automatic support for checking progress constraints, finding and notifying team members affected by progress change, identifying potential sources of progress change, and reflecting progress change in the tracking system. The process models can help team members understand the co-ordination of activities in each technical process and then adapt them to meet their needs. Therefore, the set of process models in Appendix B should be considered as representative rather than definitive.

A data model was created to store and access different types of data entity. The data model represented a wide range of data entities and the dependencies between them. These include representing tasks, stories, releases, unit tests, acceptance tests, integration tests, developers, testers, story owners and project managers. The dependencies can help identify how the development progress is affected by the technical activities and help target co-ordination support to affected team members.

The four models proposed (the version model, the user story progress model, the process model, and the data model), have together contributed in providing a strong design to the progress tracking system. The version model paid attention to the state of the source code artefacts. The user story progress model provided detailed progress information, based on the state of its corresponding source code versions. The process model successfully helped in identifying the points where a progress change takes place, reflecting this on development progress (e.g. progress states of stories), and co-ordinating the impact on the affected team members. Finally, the data model provided the infrastructure that saves the relationships between progress information (i.e. stories and their underlying tasks) and the technical objects that affect progress (i.e. source code artefacts, unit testing data, acceptance testing data and integration data).

Objective 5: *Evaluating the computer-based holistic approach. This includes preparing an evaluation methodology and determining whether the computer-based holistic approach is achievable.*

Chapter 6 evaluated the holistic approach to managing progress of distributed agile teams. The chapter explained a methodology for evaluation that relies on the scenario-based evaluation approach. It consists of three main parts: selection of scenarios, analysis of scenarios and validation of scenarios.

To achieve the first, a systematic method was used to identify suitable scenarios. This ensured that the progress change events involved in the scenarios were significant and had reasonable coverage for evaluating the holistic approach by considering the complexity degree of the progress event, frequency of the event occurrence and influence of the progress event on the development progress. In addition, scenarios were selected that included potential progress events from each technical factor. The selection process resulted in choosing three representative scenarios: Check-in Source Code Version, Performing Successful Integration and Running Automated Acceptance Testing.

In the analysis of scenarios, a comparison was made between the classical XP version of each scenario, where the holistic approach is not considered, and the new version after introducing the holistic approach. In addition, a multi-perspective view was achieved by providing a role-oriented analysis of each scenario version. The analysis showed that team members could achieve better awareness of progress with the holistic approach version of these scenarios. The holistic approach was able to provide automatic identification and co-ordination of progress change events. It immediately identified the potential change in progress resulting from the technical activity. In addition, affected team members were immediately informed about the progress change.

The validation of scenarios was achieved through developing a prototype system. The implementation successfully demonstrated that the holistic approach scenarios can be implemented with a computer-based system. The database schema and the SQL queries were able to identify the various dependencies existing in the scenarios. This helps validate the data model. In addition, the implementation of the three scenarios were able to validate the version model (i.e. moving source code versions from one state to another based on the versioning activities), the user story progress model (moving stories from one state to another based on how the technical activities affect progress) as well as the process model (i.e. providing the necessary co-ordination such as notifications).

Although the scenario-based evaluation used in this work revealed that a better awareness of progress can be achieved with the holistic approach, it is worth emphasising that a more critical assessment will be achieved if the tracking system runs in real projects and for a long time. This will help refine the system and examine the organisational impact when it is introduced.

7.2 Future Work

This section describes a number of ways in which the work presented in this thesis can be further extended.

7.2.1 Impact of Progress Change on Overall Project Plans and Velocity

The holistic approach identifies and co-ordinates potential changes that may affect development progress, but it does not tell how a recent change in progress may affect the overall iteration and release plans. A further improvement can be made by showing how the change will influence the current iteration and release. This includes identifying what tasks/stories may not be possible to carry out in the current cycle and what the new date is for providing the release to the customer.

Rather than the project manager being sent only notifications about progress changes, he may also be notified about how his plans are affected due to the change. The tracking system can also send notifications to the project manager informing him about any changes occurring to the project velocity (i.e. number of units of work completed over a period of time).

The proposed data model registers the planned time and date of each task, user story, iteration and release. In addition, from the activities provided for each technical process, calculations measuring the impact of potential changes on the iteration and the release can be made, and notifications based on that sent.

Furthermore, the proposed system can be integrated with some of the commercial agile project management (APM) tools such as Rally [14] or Mingle [15]. These tools involve professional capabilities to support making planning and re-planning activities based on the latest information about the current progress of the project. The approach created in this research provides up-to-date progress

information which can then be utilised by the APM tools to identify how changes will affect the velocity or release delivery date.

7.2.2 The Use of Change Impact Analysis Techniques

Changes to a source code artefact that directly affect its corresponding story progress are recognised easily due to the linkage between each source code artefact and tasks/stories. However, changes affecting progress could be as a result of the implicit relationship between the functionalities of one story and another.

Several approaches have been used to understand the ripple effect of one element on the other elements in the source code. One of these approaches that can be used to predict potential changes affecting progress is the *Heuristic-Based Analysis*. This approach tries to mine change history stored in the versioning control system in order to obtain useful information about change propagation required. One of the most used heuristic-based analysis techniques is the historical co-change analysis [169]. If two source code elements have been changed at the same change set, this means that they are related via a historical co-change relation. The historical co-change analysis technique is based on the following intuition: elements that changed together in the past have a high tendency to change together in the future [170]. It assumes that there is logical dependency between the co-changed elements.

The heuristic-based analysis has gained high interest in the literature recently and studies prove that it can be used to help developers in their daily work (e.g. [170] [171]). This approach can be used in this research to predict the source code artefacts potentially affected. The co-change concept can be updated in this context to mean the group of artefacts that contribute to the same task. It still serves the same purpose because tasks are normally completed by correlated artefacts.

7.3 Contribution of the Research

7.3.1 Research Publications

The author had participated in three International Conferences: 9th International Conference on Agile Processes in Software Engineering and eXtreme Programming (XP 2008, Limerick), IEEE 6th International conference on Global Software Engineering (ICGSE 2011, Helsinki), and 10th International Conference on Software Engineering Research, Management and Applications (SERA 2012, Shanghai). These conferences provided opportunities for the research to be shared with researchers and practitioners in the software engineering community, and the agile community in particular. Discussion of the research ideas and concepts with others helped refine the work presented in this thesis.

Large portions of Chapters 2, 3, 4, 5, and 6 have been published in the proceedings of the peer-reviewed ICGSE and SERA conferences and the peer-reviewed International Journal of Computer Applications. The paper published at the SERA Conference has been selected among the best papers at the conference. An extended version of this paper has been accepted for publication in the peer-reviewed Journal of Software. The list of publications are:

- S. Alyahya, WK. Ivins, WA. Gray, Co-ordination Support for Managing Progress of Distributed Agile Projects, *IEEE Sixth International Conference on Global Software Engineering-Workshop*, Helsinki, 2011.
- S. Alyahya, WK. Ivins, WA. Gray, Managing Versioning Activities to Support Tracking Progress of Distributed Agile Teams. *International Journal of Computer Applications*, February 2012. Published by Foundation of Computer Science, New York, USA.
- S. Alyahya, WK. Ivins, WA. Gray, A Holistic Approach to Developing a Progress Tracking System for Distributed Agile Teams, *ACIS 10th International Conference on Software Engineering Research, Management and Applications (SERA)*, Shanghai, 2012.

- S. Alyahya, WK. Ivins, WA. Gray, Raising the Awareness of Development Progress in Distributed Agile Projects, *Journal of Software*, Academy Publisher, Finland, Accepted.

7.3.2 Originality of the Proposed Approach

This thesis supports the effective management of progress by providing a computer-based holistic approach to managing development progress that aims to explicitly identify and co-ordinate the effects of the various technical factors on progress. It provides formal mechanisms to support the problem of progress management. This copes with the growing calls in the agile literature for more formal processes to co-ordinate distributed agile teams (e.g. [5] [172]).

Although some of the agile project management (APM) tools (e.g. Rally, TargetProcess, VersionOne) have started providing integration with the technical systems (e.g. UT tool, versioning system), these integrations are insufficient to solve the impact of technical activities on development progress. Their main purpose is to provide traceability linkages between the technical artefacts (e.g. source code artefacts and test artefacts) and the progress tracking artefacts (i.e. tasks, stories, releases). The holistic approach extends the scope of current progress tracking systems. It allows the progress tracking system to identify progress change events, not only through user inputs, but also through automatic identification. It then helps raise team members' awareness of the progress state by providing the necessary co-ordination for each progress change event. Thus, the thesis provides a step forward for agile project management tools.

Furthermore, Asklund et al. [115] mention that managing change can provide valuable information about development progress. They suggest linking each change with the tasks/stories information by adding task and story numbers as a comment with every check-in process. However, their work does not provide an approach that allows for automatic identification of potential changes that affect

development progress and does not support co-ordinating the change impact with team members. One of this work's authors, Lars Bendix from Lund University (Sweden), along with another author, Christian Pendleton from agile consultancy company called SoftHouse, recently use our work as an example of implementing processes to support managing changes that occur to project artefacts (source code artefacts, tests, etc) in distributed agile projects and to support providing awareness about the state of these artefacts [173] (August 2012).

The author believes that new knowledge has been added to the field by providing the holistic approach to managing development progress of distributed agile teams. The holistic approach will help distributed team members become aware of the actual progress of the project. They will be able to know the state of the project artefacts that they use. They will also be made aware of potential changes, affecting progress, to the items that they are responsible for as soon as these changes occur. By doing so, the approach can support solving the problem of having large acceptance tests failing at the end of each iteration and release, due to the lack of good progress tracking mechanisms.

The approach is achieved through identifying the co-ordination support necessary for managing progress change events and designing a computer-based system capable of providing the necessary co-ordination. 23 progress change events, caused by the technical activities, were identified along with identification of the co-ordination support required for each progress change event. In addition, the four models proposed have contributed to providing a strong design of the progress tracking system. These models are: the version model, the user story progress model, the process model and the data model.

Appendix A

Agile Principles

The principles behind the Agile Manifesto are [12]*:*

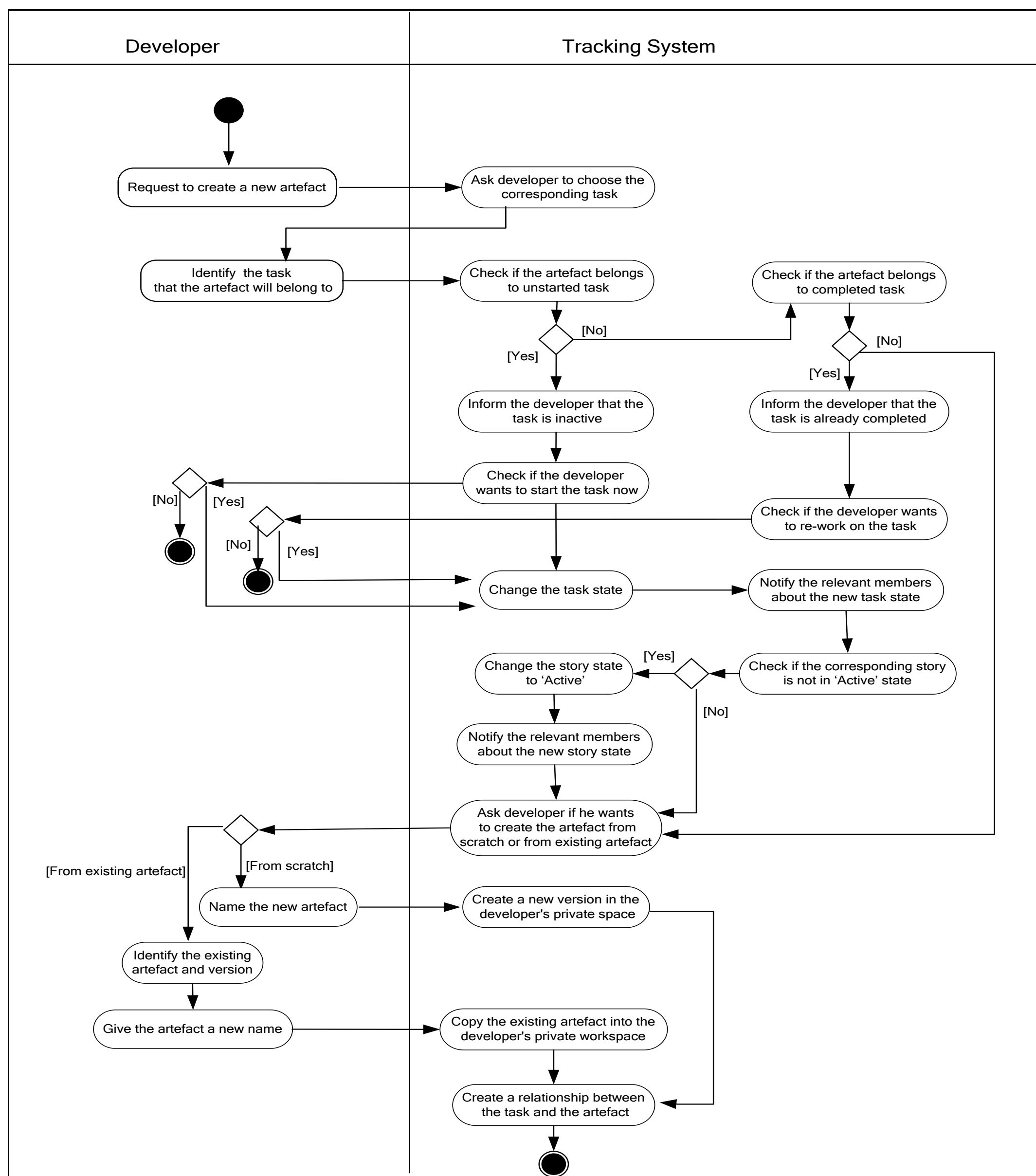
1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of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

Appendix B

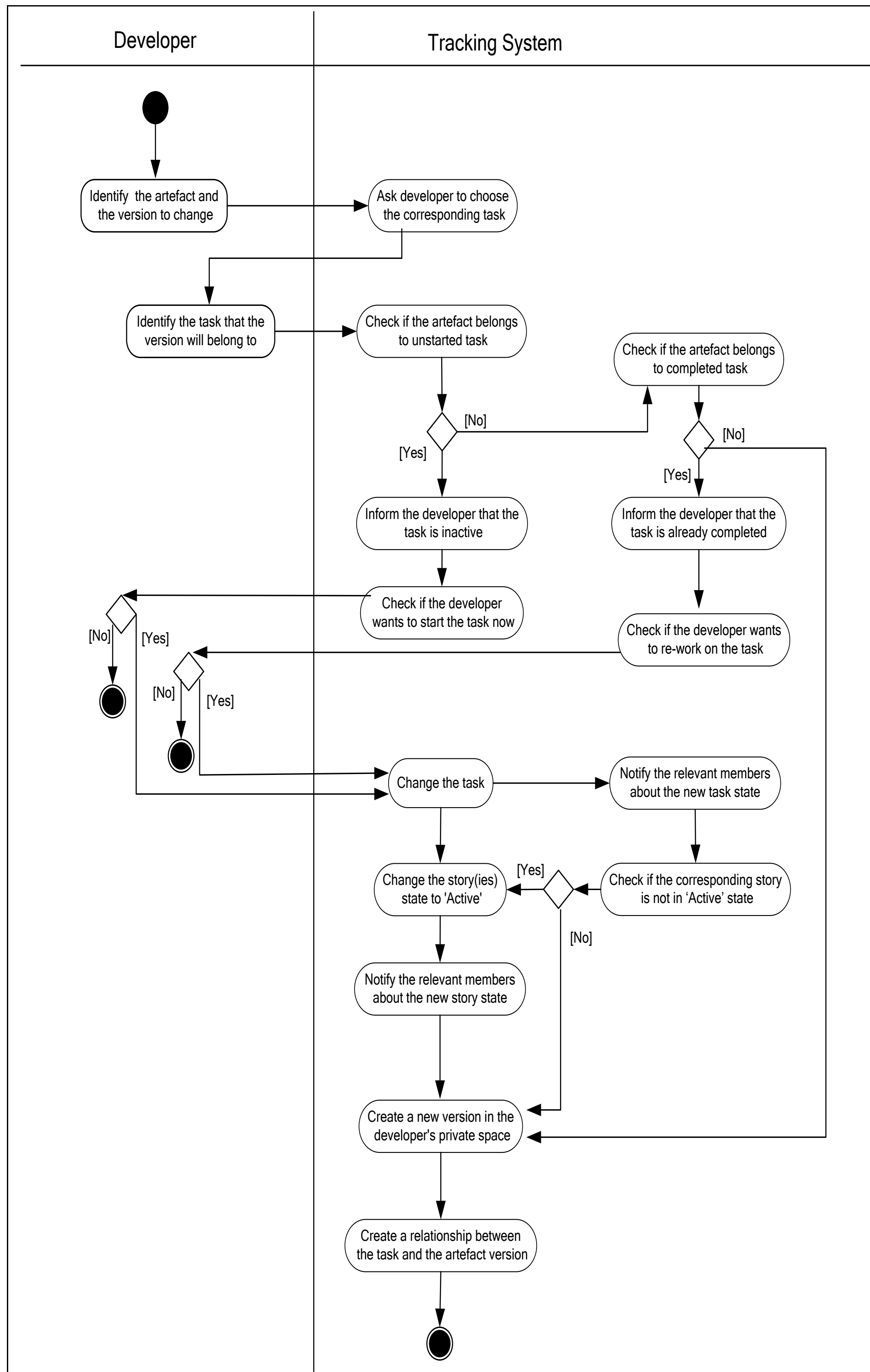
Technical Process Models

Source Code Versioning Process Models

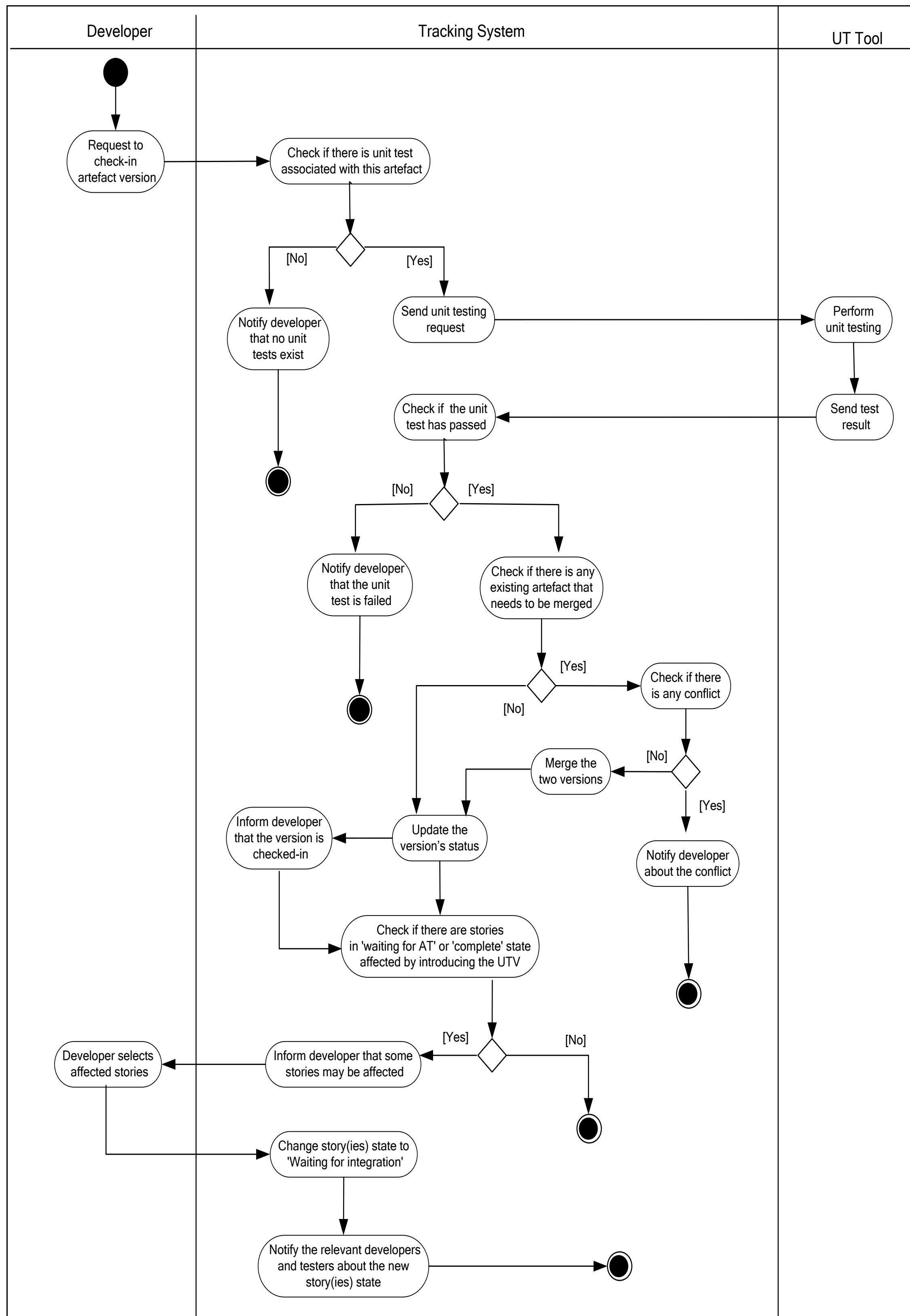
Create a new source code artefact:



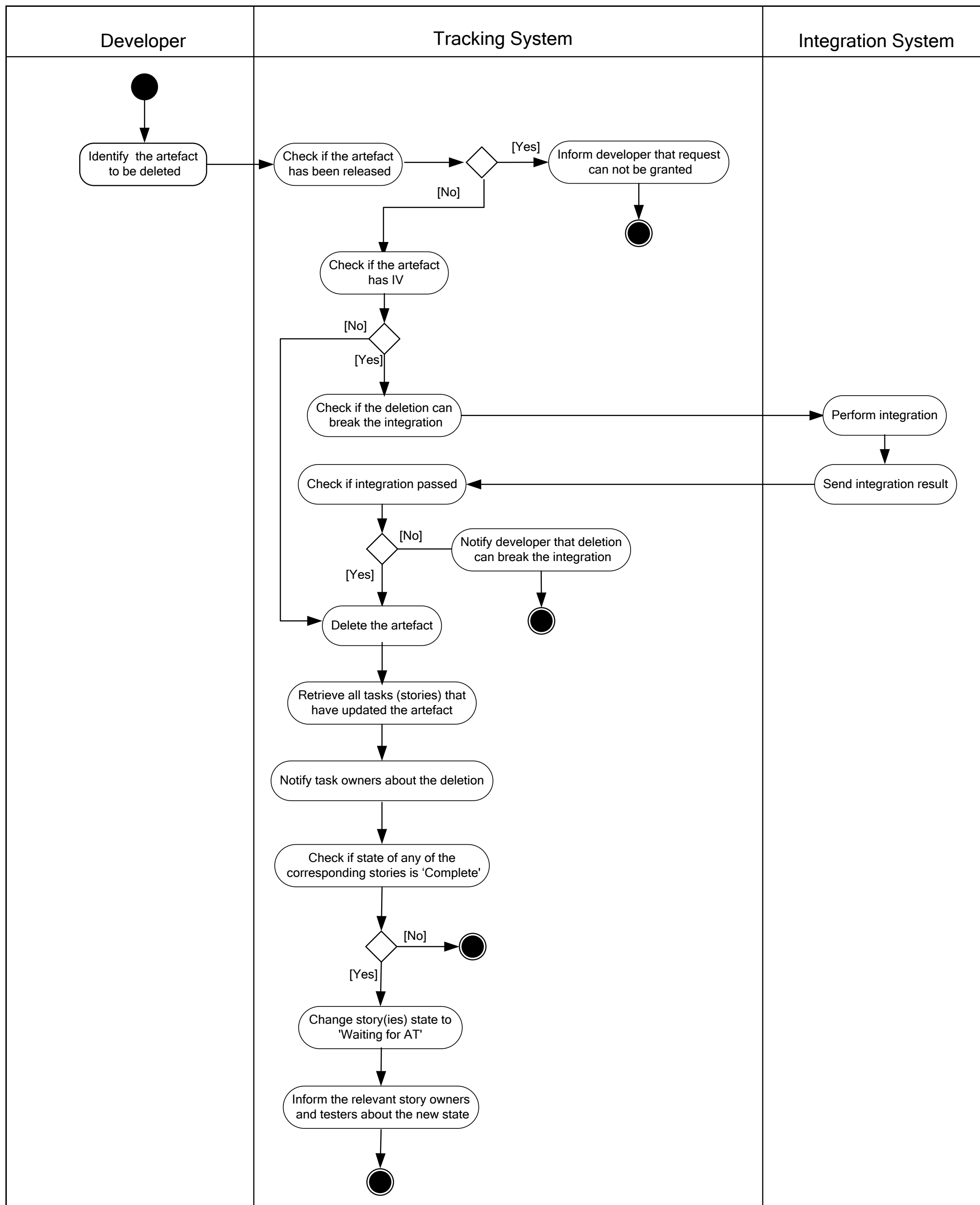
Check-out a source code version:



Check-in a source code version:

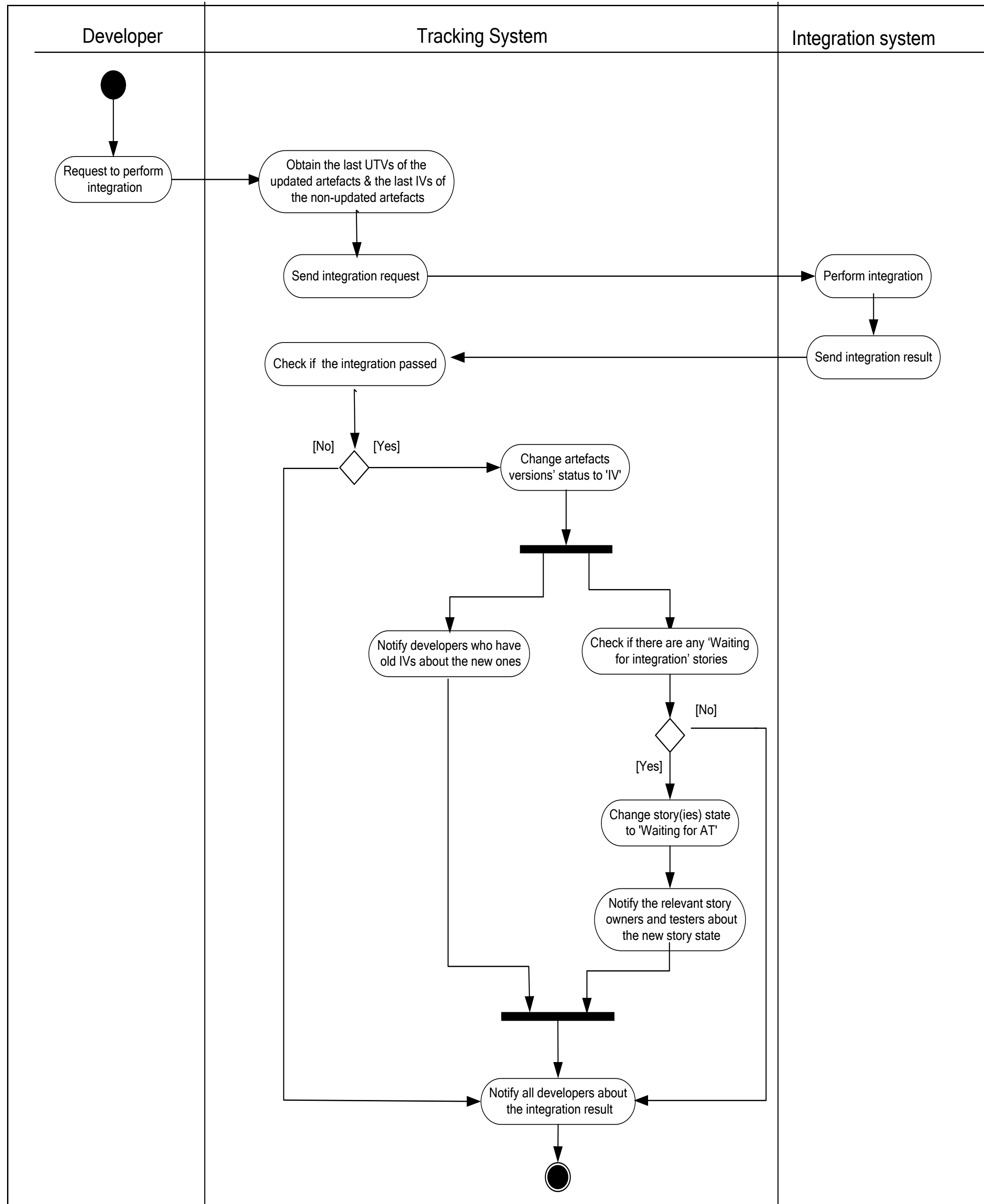


Delete an artefact:

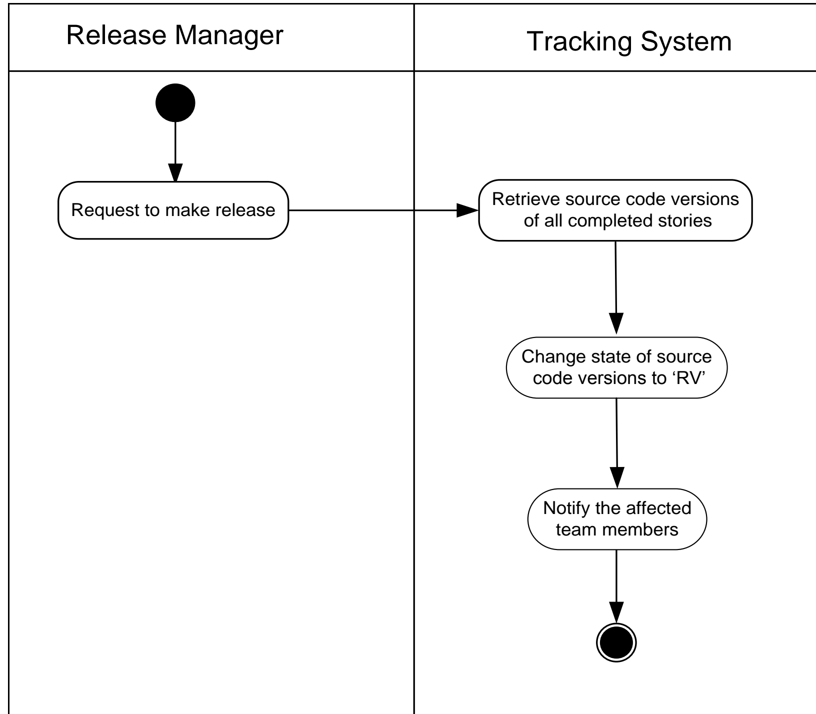


CI and Releasing Process Models

Perform integration:

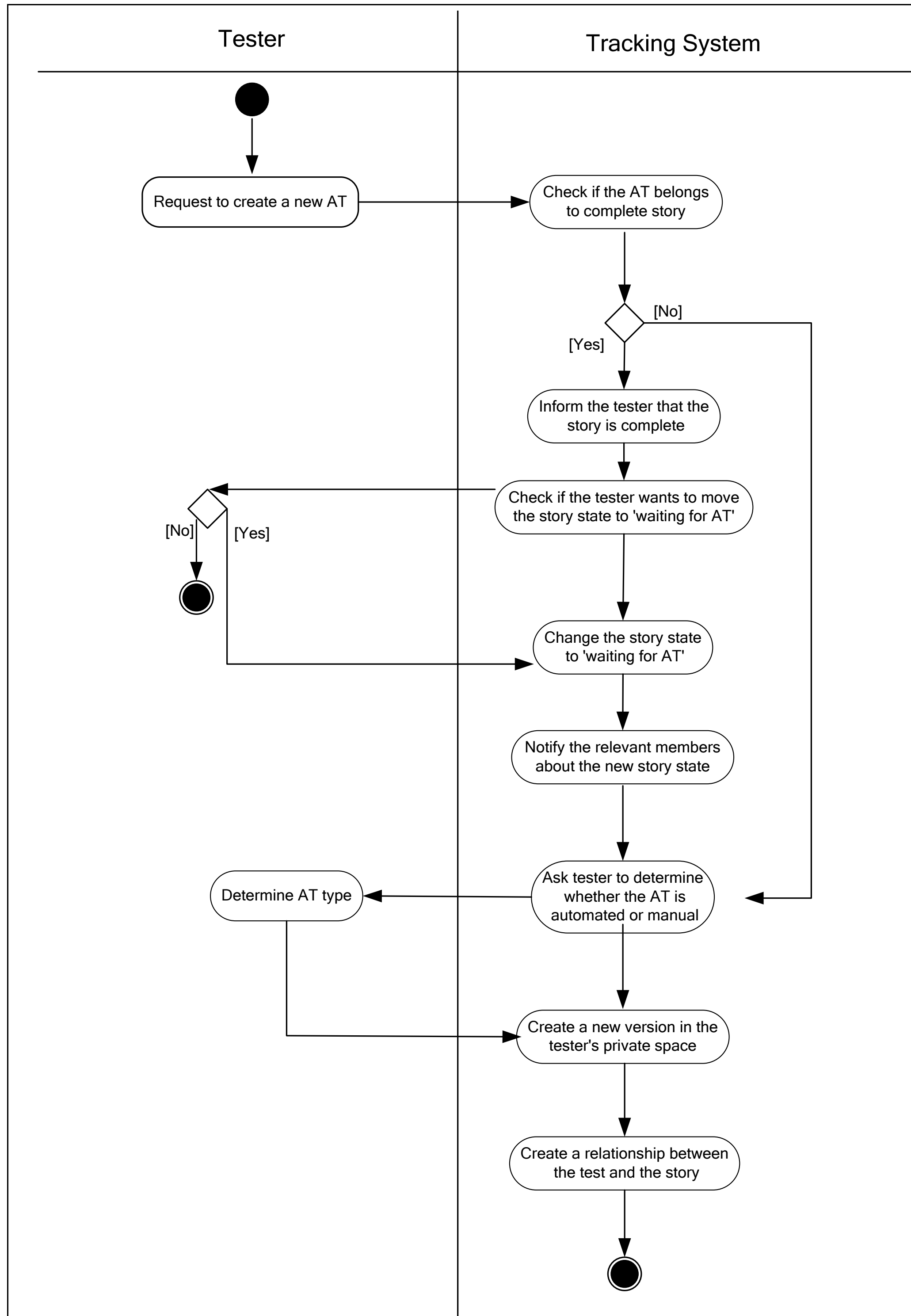


Make a release:

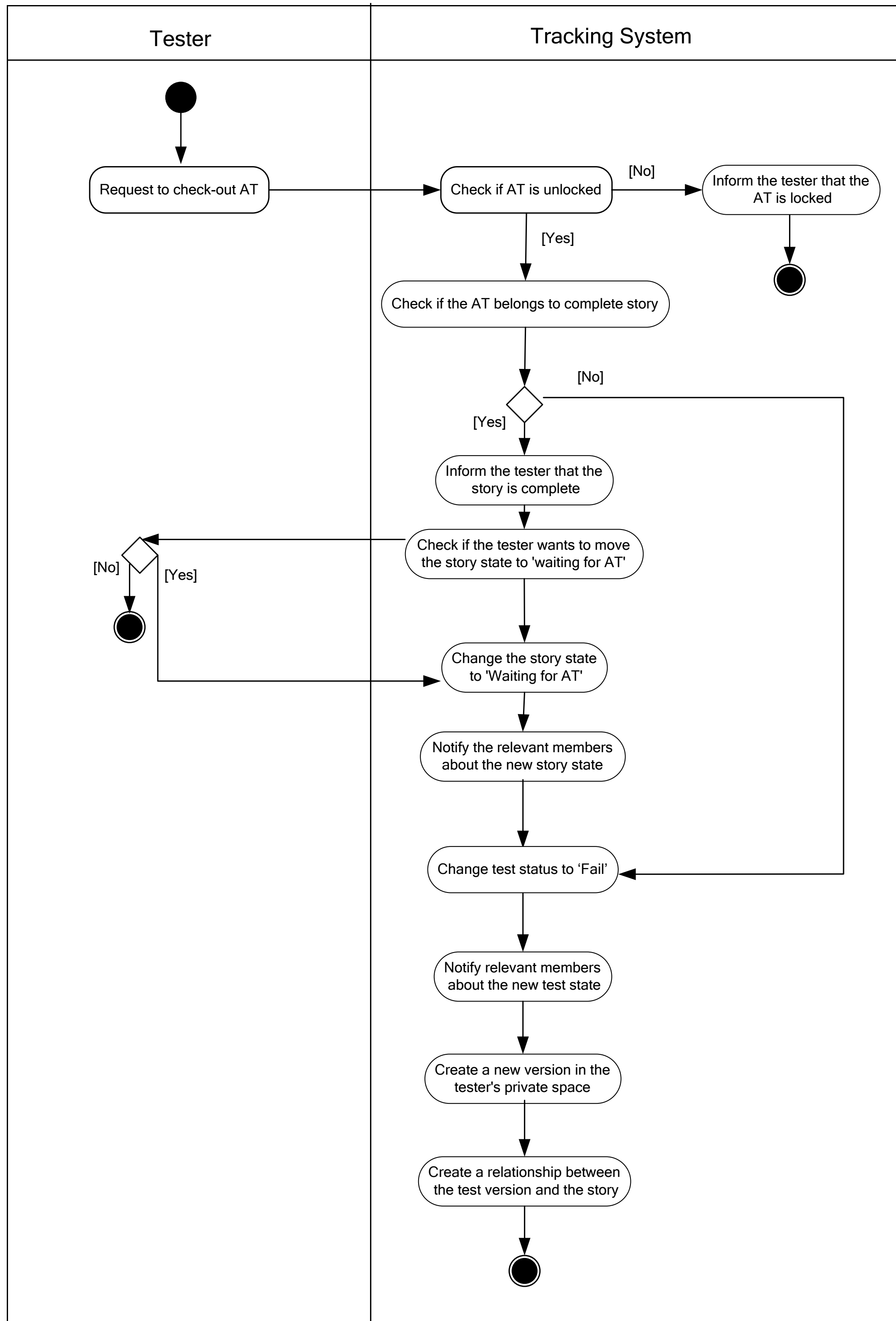


AT Process Models

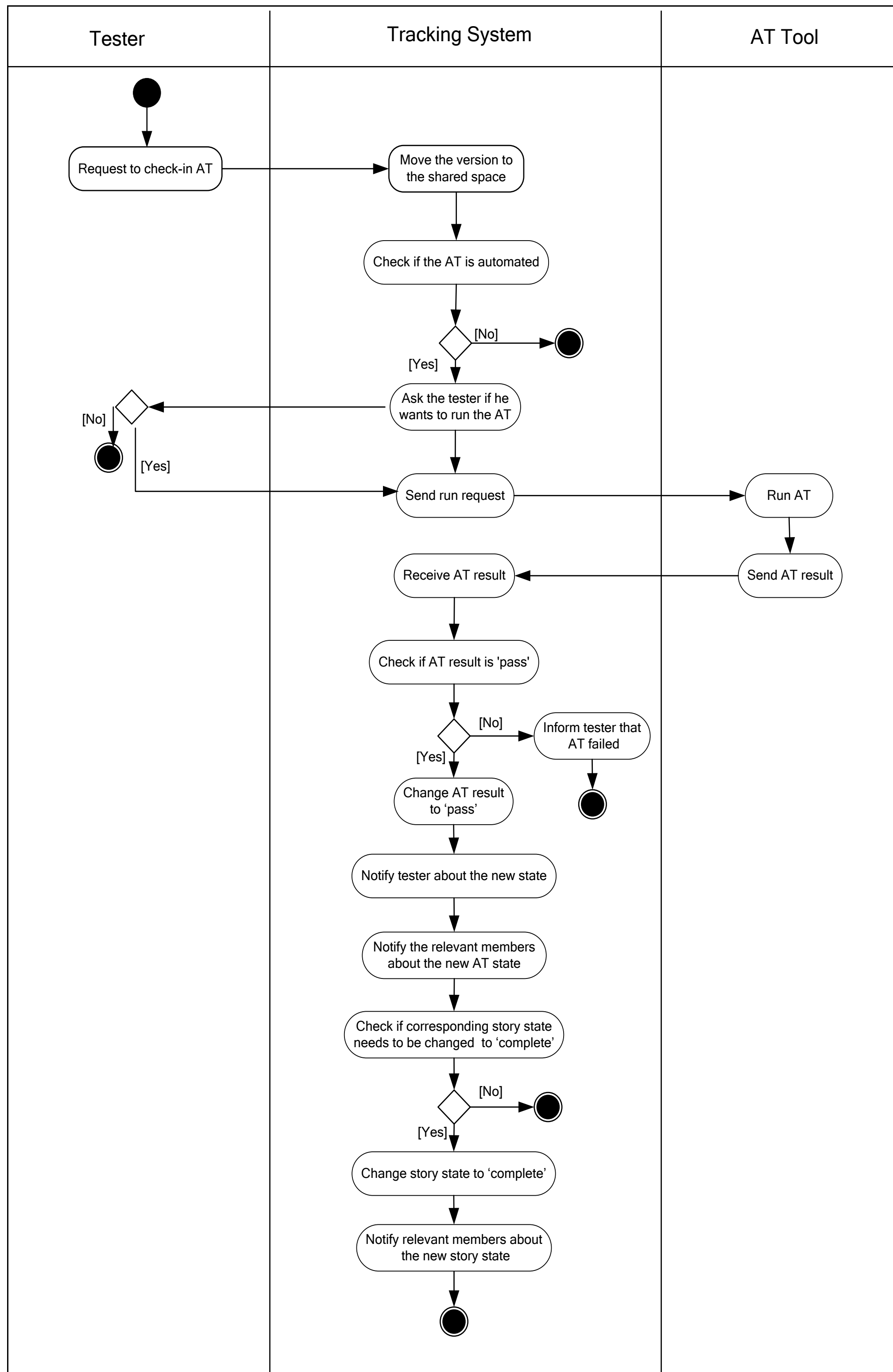
Create a new acceptance test:



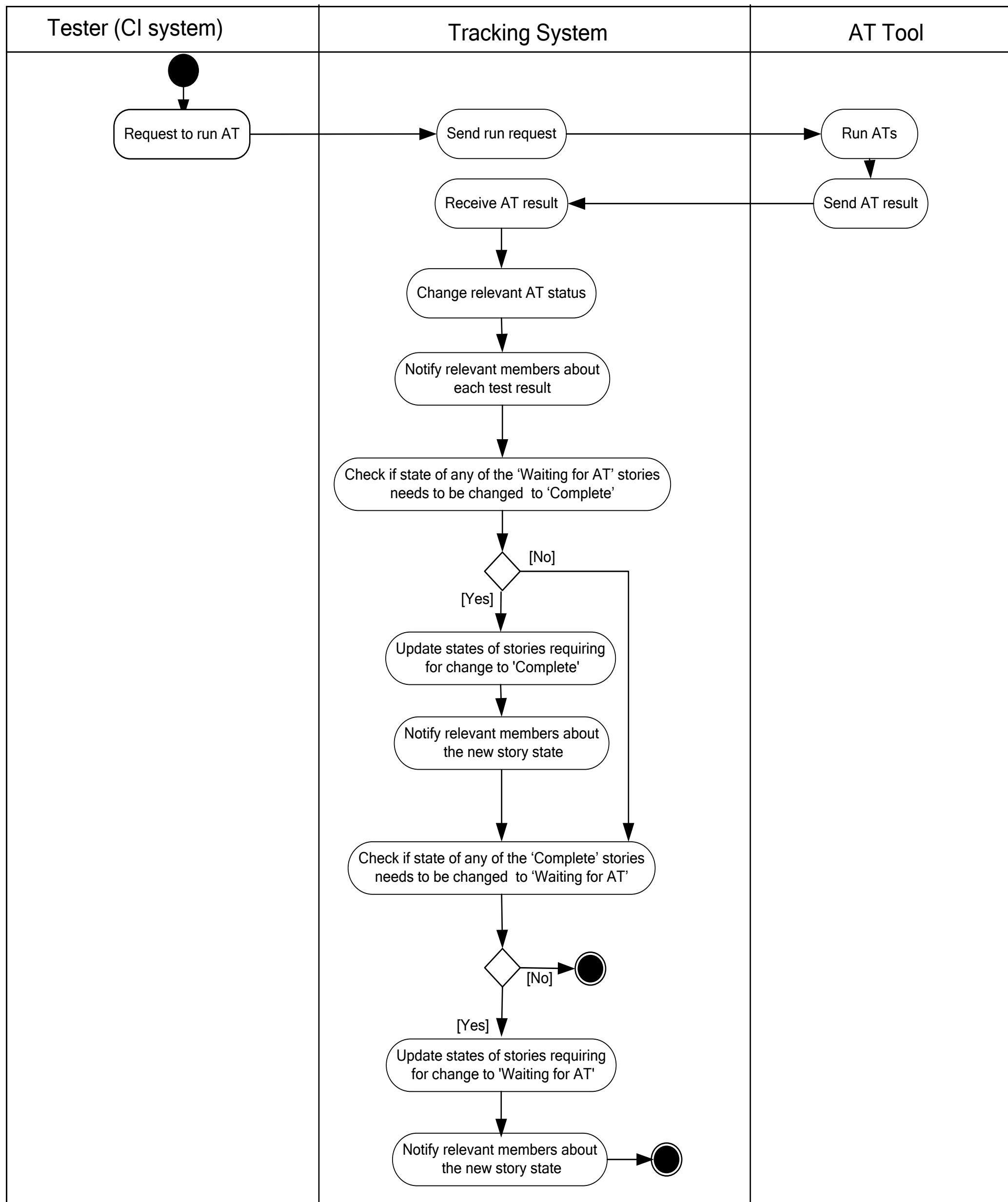
Check-out acceptance test version:



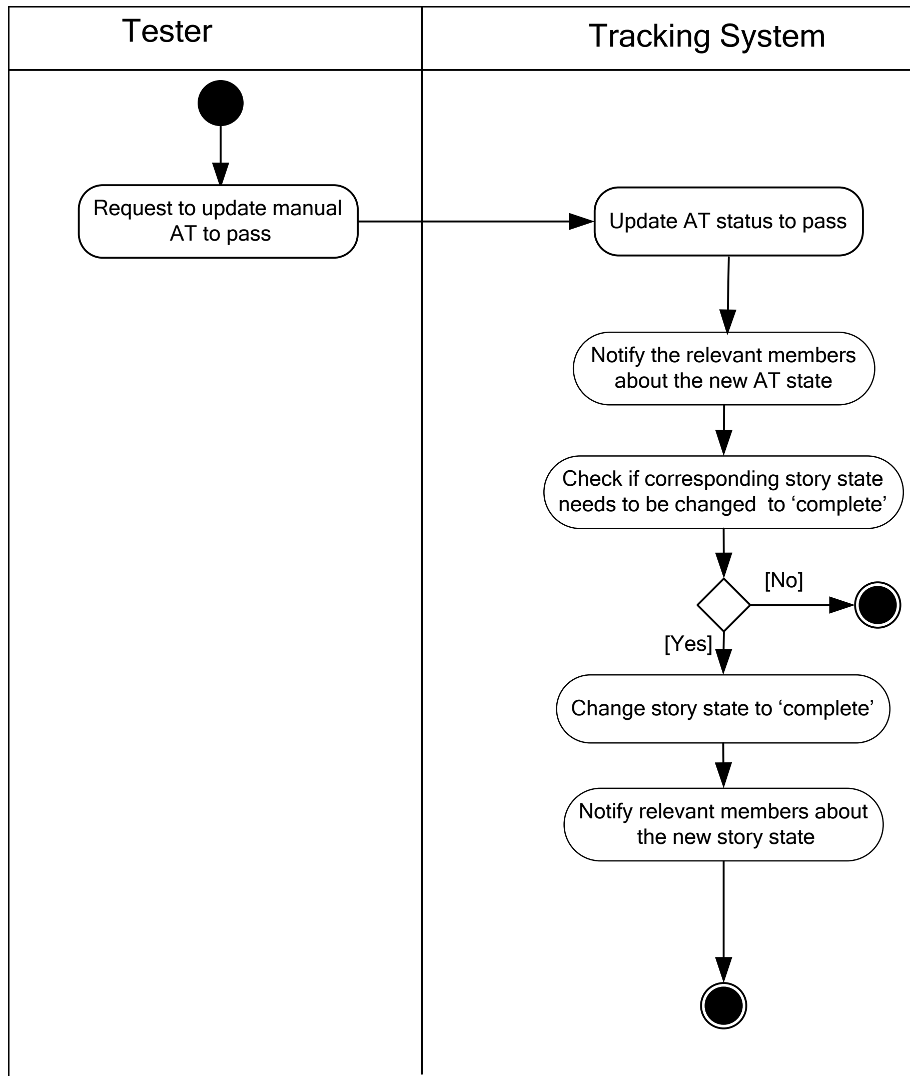
Check-in acceptance test



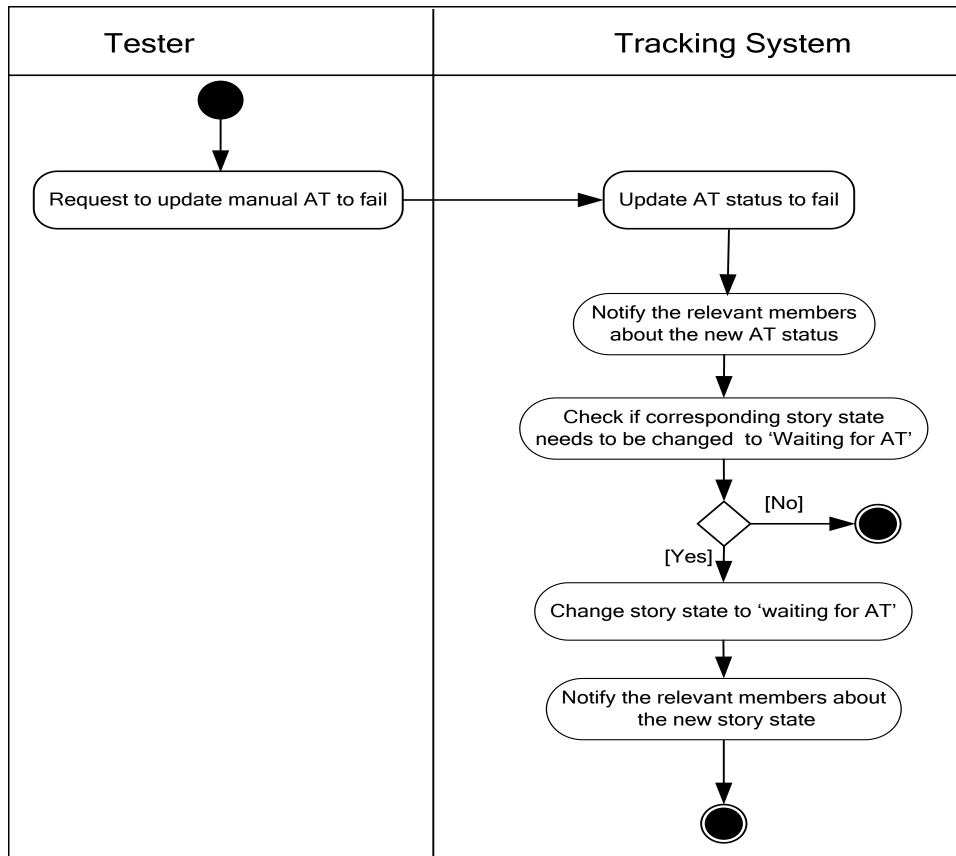
Run acceptance testing:



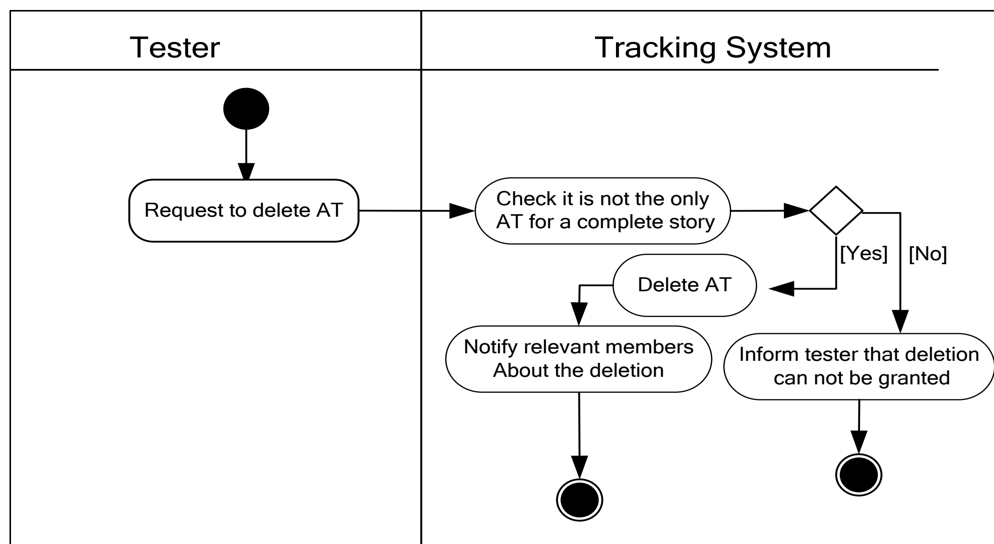
Update manual acceptance test to pass:



Update manual acceptance test to fail:

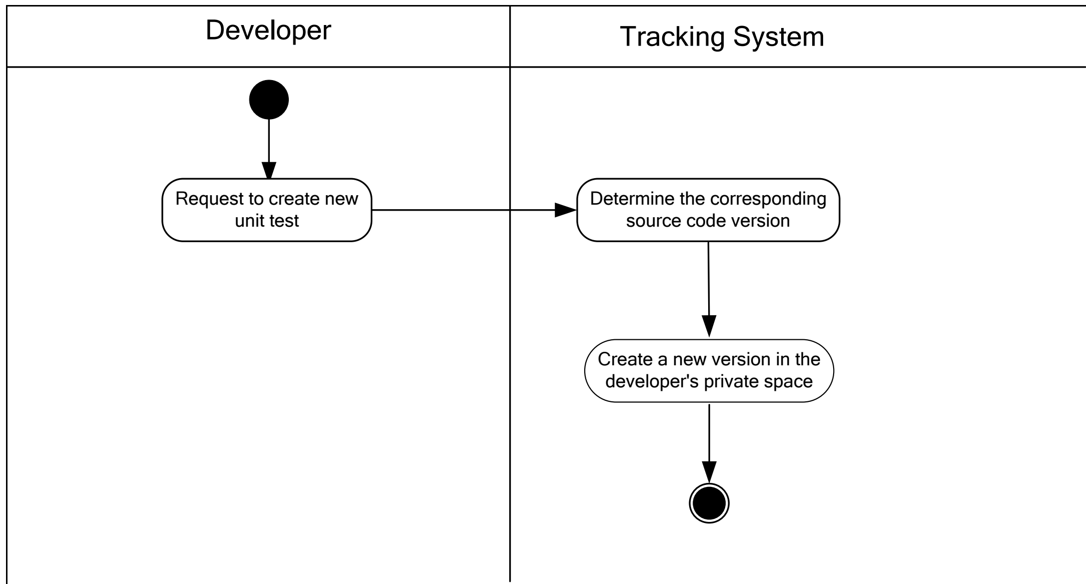


Delete an acceptance test:

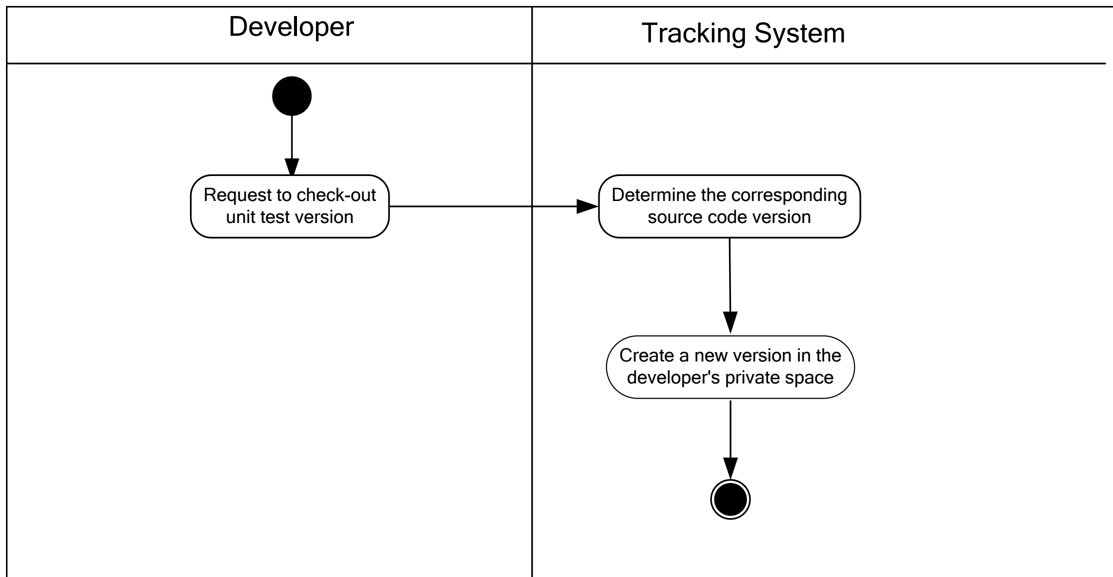


UT Process Models

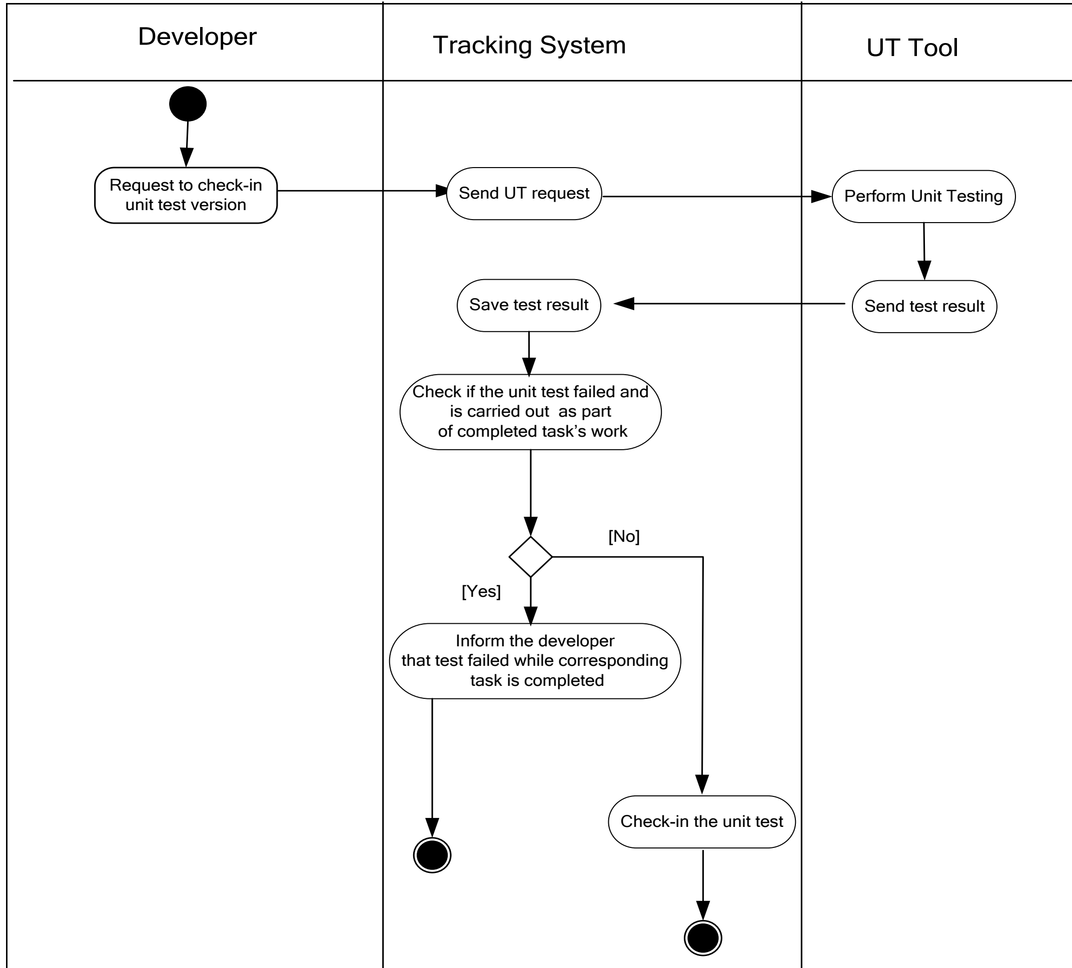
Create a new unit test:



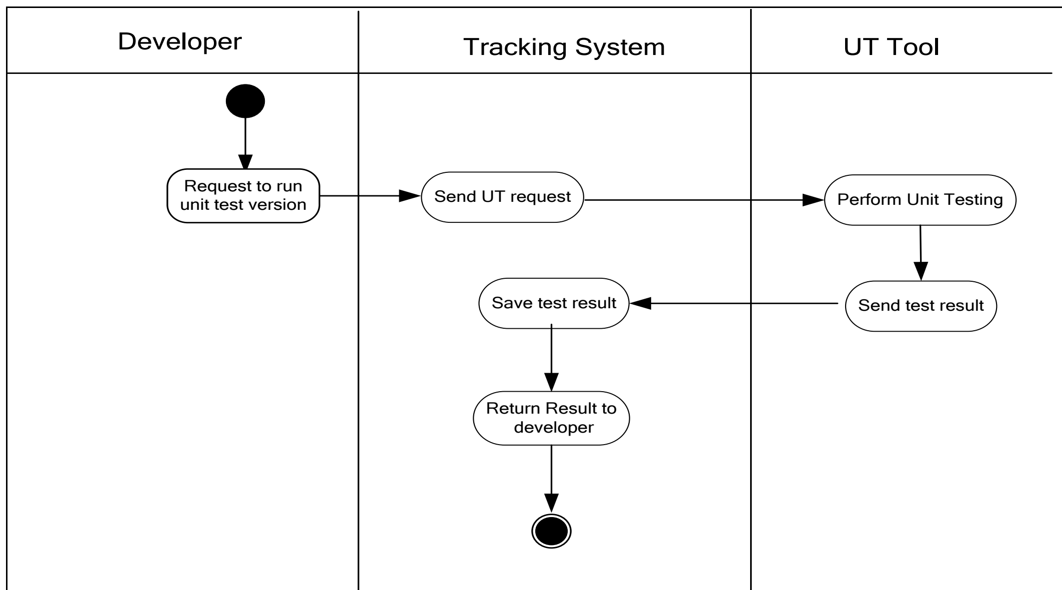
Check-out unit test version:



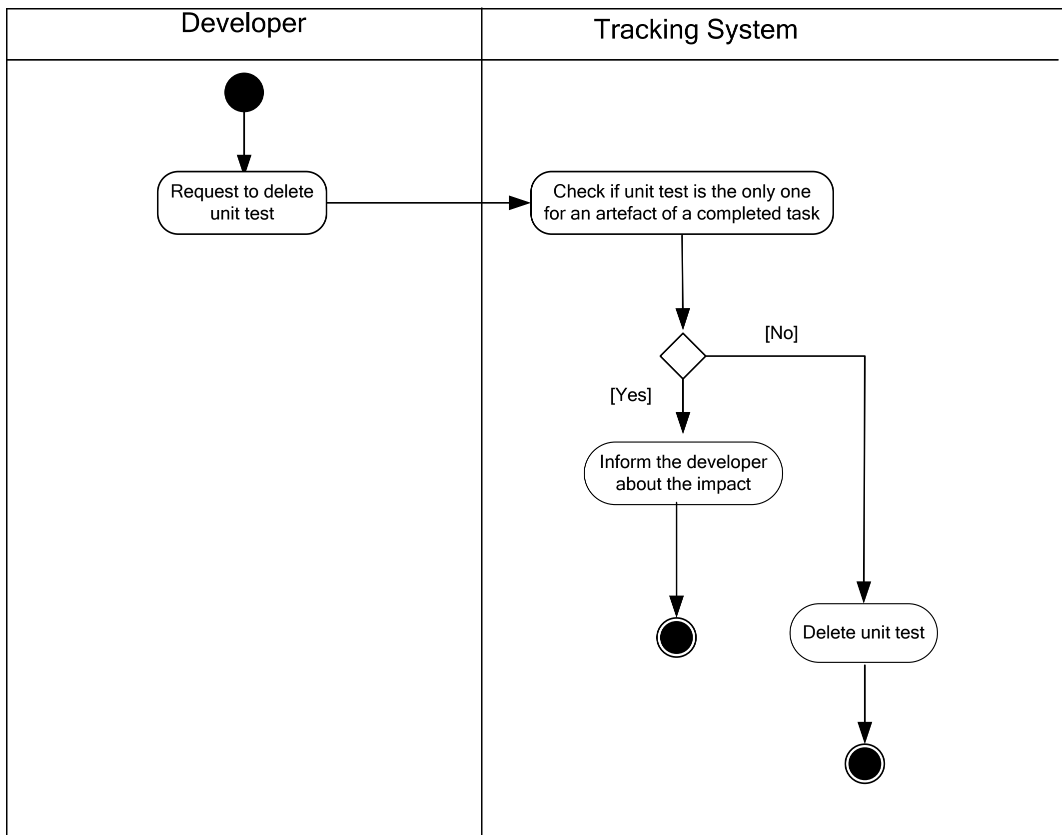
Check-in unit test version:



Run unit test:



Delete unit test:



Appendix C

Implementation Description for the Holistic Approach Version of Scenarios 2 and 3

Implementation of Scenario 2 (Performing Successful Integration)

The scenario starts with the following initial data set:

- The functionalities required for user story US1 have been implemented, and the story is waiting for its new versions A5.1 and A6.1 to be integrated.

- The functionalities required for user story US2 have been implemented, and the story is waiting for its new versions A8.1 and A9.1 to be integrated.

- The user story US3 is still active and has the following new versions entering the integration process: A12.1, A13.2 and A14.2.

- The user story US4 is still active and has the following new versions entering the integration process: A15.1, A16.1 and A18.1.

The implementation of the five steps involved in the holistic approach version of the integration scenario is discussed below.

1- Sally clicks on 'Perform Integration' in the tracking system (Figure A3-1).

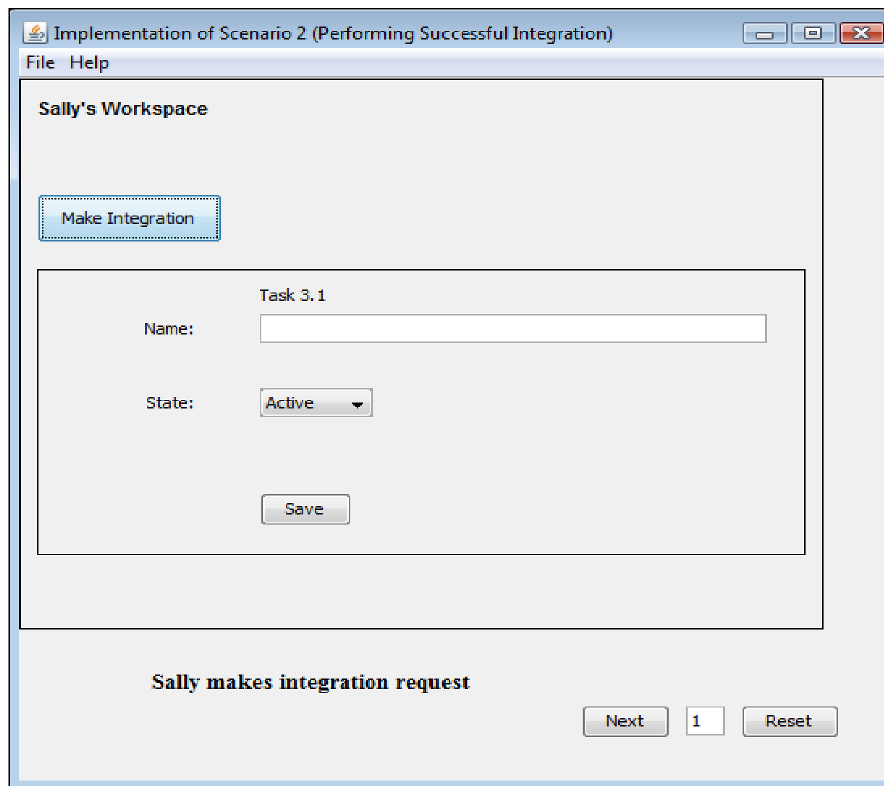


Figure A3-1. Implementation of scenario 2, step 1.

The new integration process is registered in the database through the following SQL query:

```
INSERT
INTO integration(result,creator)
VALUES ('In Progress','Sally')
```

2- The system retrieves the last UTVs of the recently updated artefacts and the last IVs of the non-recently updated artefacts and sends an integration request to the continuous integration (CI) system.

The 'DevelopmentArtefact' table can help determine the last UTV version and the last IV version for each source code artefact. Hence, in order to retrieve the required versions, we need to compare the timestamps of those two versions for each artefact.

If the last UTV version is more recent than the last IV version for an artefact, this means that the artefact has been updated since it was last integrated; this requires the recent UTV version to be integrated. Otherwise, the IV version is chosen to enter the integration process.

In the case of having an artefact without the IV version, the UTV version is selected, as this means that the artefact has not entered any integration process thus far.

The selected versions are kept in the table 'VersionIntegration', which shows which versions entered in which integration processes. The code overleaf is used to perform the second step in the scenario. We update the timestamps of the last UTVs and IVs several times to ensure that the code satisfies the various cases.

```
// the columens:id, lastUT and LastIV of the table 'DevelopmentArtefact' are stored in
a table model 'tm1'
tm1=  jTable1.getModel();
int s= developmentartefactList.size();
int i;
for(i=0;i<s;i++) // Each cycle compares the last UTV and the last IV of an artefact
{String lastutv = tm1.getValueAt(i, 1).toString();
String lastiv = tm1.getValueAt(i, 2).toString();
request1 = java.beans.Beans.isDesignTime() ? null :
scenario2PUEntityManager.createNativeQuery("select Timestamp from developmentversion
where versionid='"+lastutv+"'"); //retrieving the timestamp of the last UTV
request2 = java.beans.Beans.isDesignTime() ? null :
scenario2PUEntityManager.createNativeQuery("select Timestamp from developmentversion
where versionid='"+lastiv+"'"); //retrieving the timestamp of the last IV
String timee = request1.getResultList().toString();
String timeee = request2.getResultList().toString();
if ( !(request1.getResultList().isEmpty()))
{ if ( !(request2.getResultList().isEmpty()))
{ // It is required first to remove the brackets from the received queries
timee= timee.substring(2, 23);
timeee= timeee.substring(2, 23);
Timestamp ts1= Timestamp.valueOf(timee);
Timestamp ts2= Timestamp.valueOf(timeee);

if (ts1.after(ts2)) // if the timestamp of the UTV version is more recent than the IV
version
{ jTextArea1.append(" value of object is: "+lastutv + " State: UTV, Time: " +
ts1.toString()+"\n");
scenario2PUEntityManager.getTransaction().begin();
request3 = java.beans.Beans.isDesignTime() ? null :
scenario2PUEntityManager.createNativeQuery("INSERT INTO
versionintegration(IntegrationID,Version)  VALUES ('"+id+"','"+lastutv+"')"); // the
id here and in the following insert statements is the integration id
 int rowCount2 = request3.executeUpdate();
scenario2PUEntityManager.flush();
scenario2PUEntityManager.getTransaction().commit();
}
else // if the timestamp of the UTV version is not more recent than the IV version
{ scenario2PUEntityManager.getTransaction().begin();
request3 = java.beans.Beans.isDesignTime() ? null :
scenario2PUEntityManager.createNativeQuery("INSERT INTO
versionintegration(IntegrationID,Version)  VALUES ('"+id+"','"+lastiv+"')");
int rowCount2 = request3.executeUpdate();
scenario2PUEntityManager.flush();
scenario2PUEntityManager.getTransaction().commit();
```

```
else // There is no IV version
{
timee= timee.substring(2, 23); // to remove the brackets from the received query
Timestamp ts1= Timestamp.valueOf(timee);
jTextArea1.append (" value of object is: "+lastutv + " State: UTV, Time: " +
ts1.toString()+"\n");
scenario2PUEntityManager.getTransaction().begin();
request3 = java.beans.Beans.isDesignTime() ? null :
scenario2PUEntityManager.createNativeQuery("INSERT INTO
versionintegration(IntegrationID,Version)  VALUES ('"+id+"','"+lastutv+"')");
int rowCount2 = request3.executeUpdate();
scenario2PUEntityManager.flush();
scenario2PUEntityManager.getTransaction().commit();
}
}
else if ( !(request2.getResultList().isEmpty()))
{
timeee= timeee.substring(2, 23); // to remove the brackets from the received query
Timestamp ts2= Timestamp.valueOf(timeee);
jTextArea1.append ("value of object is: "+lastiv + " State: IV, Ttime: " +
ts2.toString()+"\n");
scenario2PUEntityManager.getTransaction().begin();
request3 = java.beans.Beans.isDesignTime() ? null :
scenario2PUEntityManager.createNativeQuery("INSERT INTO
versionintegration(IntegrationID,Version)  VALUES ('"+id+"','"+lastiv+"')");
int rowCount2 = request3.executeUpdate();
scenario2PUEntityManager.flush();
```

Implementing the previous code results in retrieving the versions displayed in Figure A3-2. For each version, the following information is displayed: version ID, Version State and Version Timestamp.

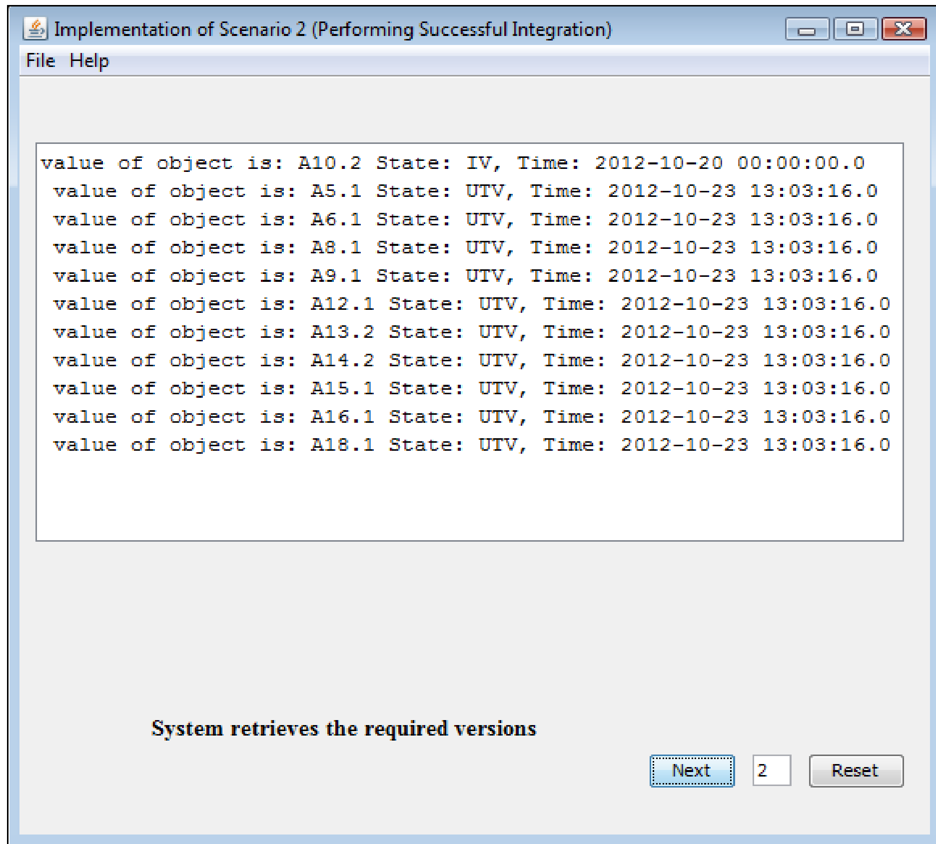


Figure A3-2. Implementation of scenario 2, step 2.

3- The system receives ‘Successful’ result from the CI system and updates the UTV versions to ‘IV’.

Because the scenario post-conditions show that the integration is successful, the integration result is be stored in the database as 'pass' using the following query:

```
"Update integration
Set result= 'Pass'
where id='"+id+"'"
```

The second ‘id’ shown in italics in the last line of the query refers to a variable identified in the code that stores the integration id of the current integration process.

A successful integration process requires updating the state of the involved UTV versions to 'IV'. This is done using the following Update statement:

```
"Update developmentversion
Set vstate= 'IV'
Where versionid in (Select version
                    From versionintegration
                    Where integrationid='"+id+"')"
```

4,5. The system checks if there are any ‘Waiting for Integration’ user stories. It moves the stories US1 and US2 to ‘Waiting for AT’. In addition to the generic notifications sent to all team members about the integration result, specialised notifications, clarifying the new state of US1 and US2, are sent to those responsible for US1 and US2, story owners (Steve and Ahmed) and testers (James and Sara) as well as the project manager.

The stories that need to be moved to 'Waiting for Acceptance Testing' are retrieved through the following query:

```
"Select  Distinct s.id
From story s, task t, developmentversion dv
Where (s.state='Waiting for Integration') and
      (t.storyid= s.id) and(dv.taskid= t.id) and
       dv.versionid in (Select version
                        From versionintegration
                        Where integrationid='"+id+"')"
```

For each of the affected stories, the story owners and testers are retrieved through the following two queries:

```
"Select so.name
From story s, storyowner so
Where s.ownerid=so.id and s.id='"+story+"'"
```

```
"Select t.name
From story s, tester t
Where s.testerid=t.id and s.id='"+story+"'"
```

The ‘story’ symbol shown in italics in the previous two queries refers to a variable in the code that represents an affected story's id.

The affected user stories, story owners and testers are shown in Figure A3-3.

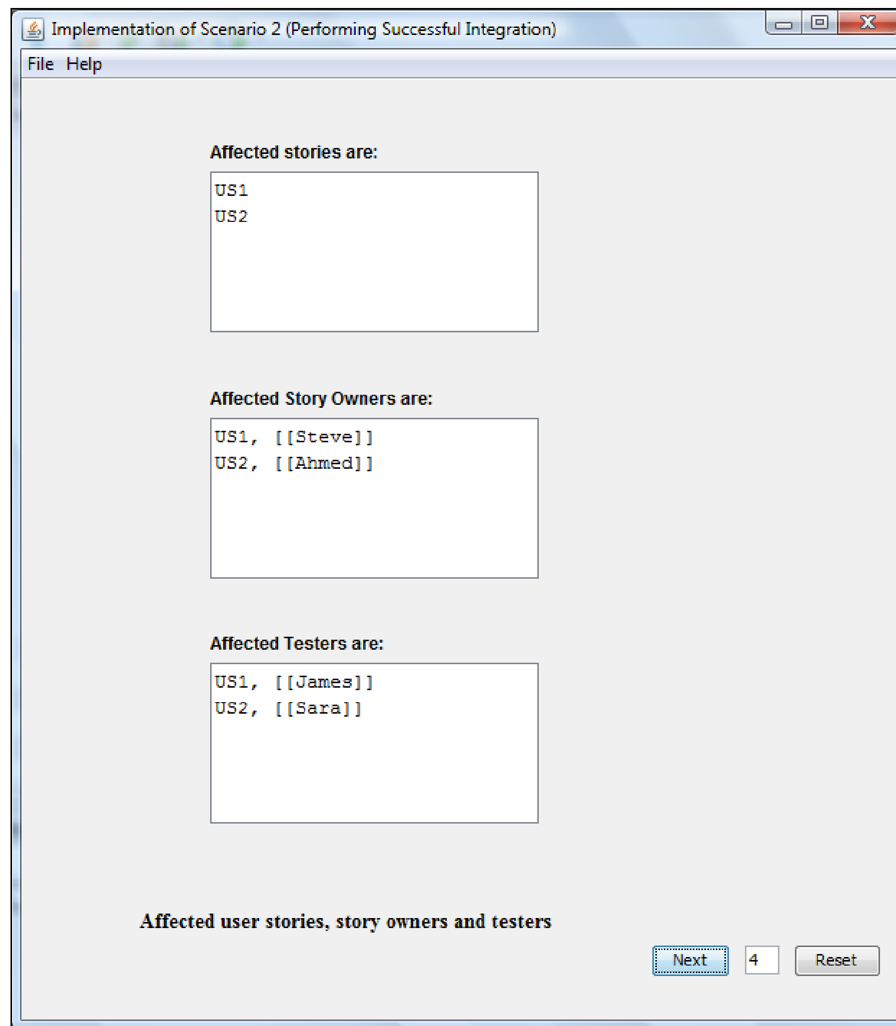


Figure A3-3. Implementation of scenario 2, steps 4.

A notification message is sent automatically to each of the affected team members. Example of such message is shown in Figure A3-4. It shows a notification message sent to Steve, the story owner of US1.

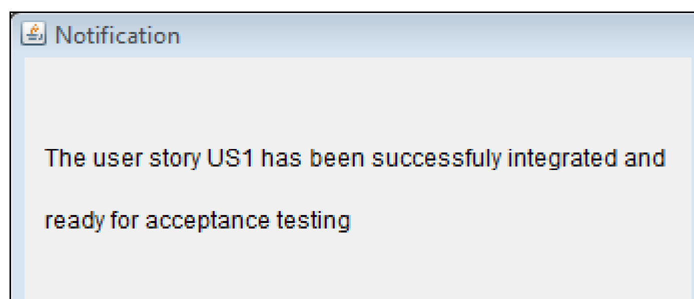


Figure A3-4. Implementation of scenario 2, step 5.

Implementation of Scenario 3 (Running Automated Acceptance Testing)

Scenario 3 starts with the following initial data set:

- The acceptance tests AT1.1, AT2.1 and AT3.1 have passed.

- These tests belong to the completed stories US1, US2 and US3, respectively.

Implementation of the four steps involved in the holistic version of the integration scenario is discussed below.

1- Sam initiates automated acceptance testing using the tracking system.

The current states of AT1.1, AT2.1 and AT3.1 are displayed in Figure A3-5.

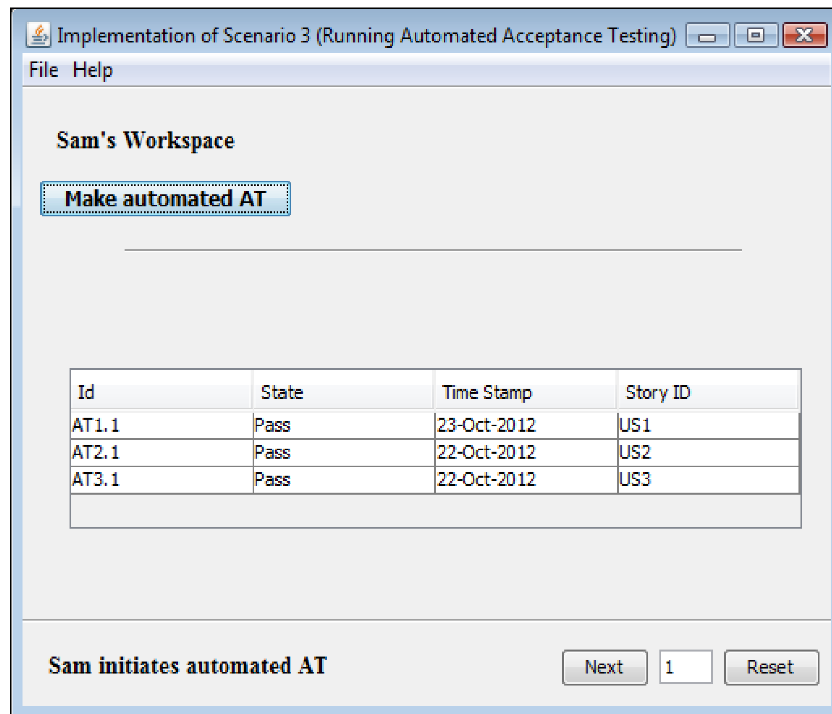


Figure A3-5. Implementation of scenario 3, step 1.

2- The tracking system sends a request to the AT tool and then receives the test result.

The test results show that acceptance tests AT1.1 and AT2.1 have failed. Hence, the state of each test needs to be updated in the database. This can be achieved through the following SQL query:

```
"Update ATversion
Set state='Fail'
Where id='"+ s[i]+ "'"
```

The 's[i]' symbol shown in italics in the previous query refers to an array in the source code that stores the id value of the failed acceptance tests.

3,4. The tracking system changes the state of user stories US1 and US2 to 'Waiting for AT'. The tracking system provides the result to Sam and automatically sends notifications to the affected team members.

The stories that need to be moved to 'Waiting for Acceptance Testing' are retrieved through the following query:

```
"Select storyid
From atversion
Where id='"+failedtests[i]+"'"
```

Similar to the previous scenario, for each of the affected stories, the story owners and testers are retrieved through the following two queries:

```
"Select so.name
From story s, storyowner so
Where s.ownerid=so.id and
      s.id='"+story+"'"
```

```
"Select t.name
From story s, tester t
Where s.testerid=t.id and
      s.id='"+story+"'"
```

The 'story' symbol shown in italics in the previous two queries refers to a variable in the source code that represents an affected story's id.

The affected user stories, story owners and testers are shown in Figure A3-6.

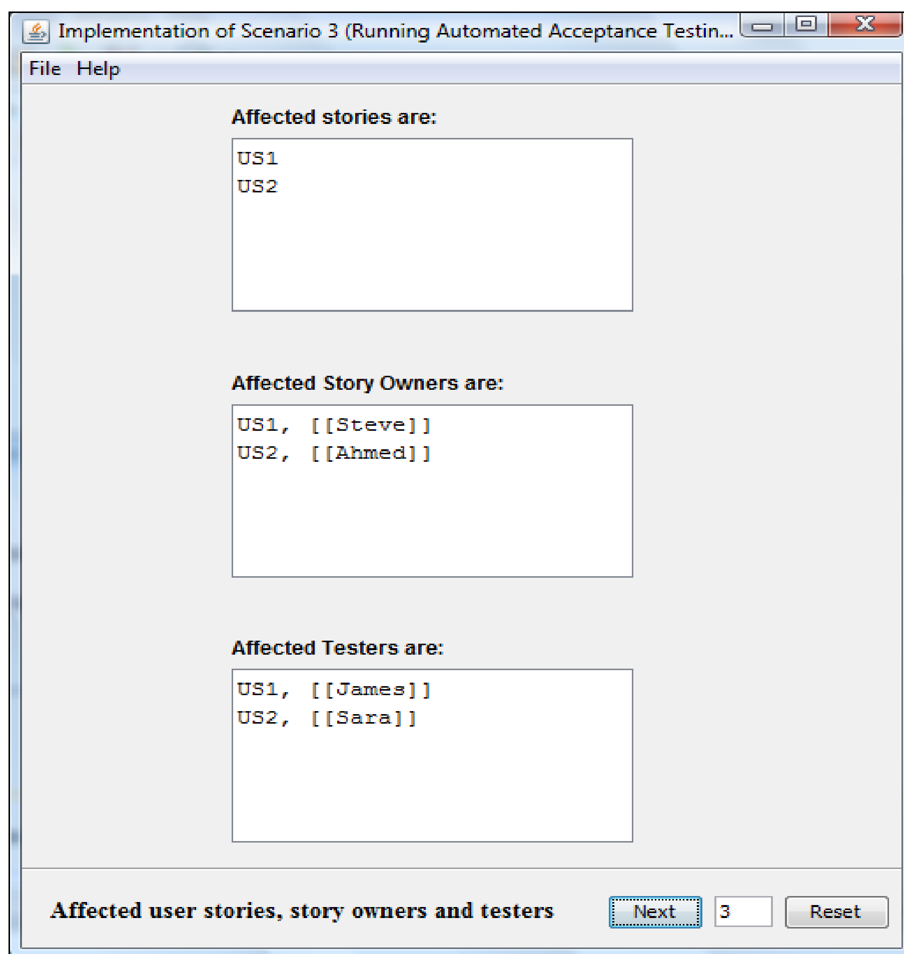


Figure A3-6. Implementation of scenario 3, step 3.

A notification message is sent automatically to each of the affected team members. Example of such message is shown in Figure A3-7. It shows a notification message sent to Sara, the tester responsible for US2.

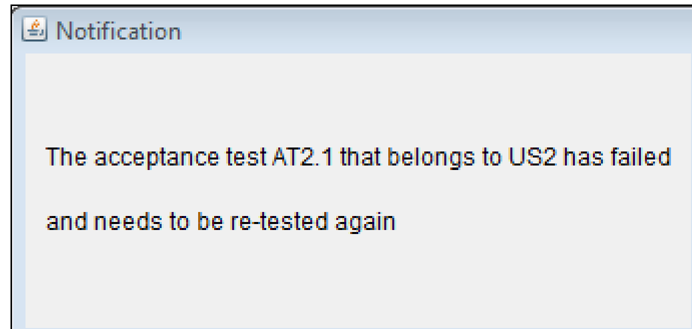


Figure A3-7. Implementation of scenario 3, step 4.

Bibliography

[1] J. D. Herbsleb, "Global Software Engineering: The Future of Socio-technical Coordination," in *FOSE '07 Future of Software Engineering*, 2007, vol. 24, pp. 188–198.

[2] J. Noll, S. Beecham, and I. Richardson, "Global software development and collaboration: barriers and solutions," *ACM Inroads*, vol. 1, no. 3, pp. 66–78, 2010.

[3] E. Carmel, *Global software teams: collaborating across borders and time zones*, vol. ISBN:0–13-. Prentice Hall PTR, 1999, p. 269.

[4] J. D. Herbsleb and D. Moitra, "Global software development," *IEEE Software*, vol. 18, no. 2, pp. 16–20, 2001.

[5] B. Ramesh, L. Cao, K. Mohan, and P. Xu, "Can distributed software development be agile?," *Communications of the ACM*, vol. 49, no. 10, pp. 41–46, 2006.

[6] J. Sutherland, G. Schoonheim, N. Kumar, V. Pandey, and S. Vishal, "Fully Distributed Scrum: Linear Scalability of Production between San Francisco and India," in *2009 Agile Conference*, 2009, pp. 277–282.

[7] J. Sutherland, G. Schoonheim, and M. Rijk, "Fully Distributed Scrum: Replicating Local Productivity and Quality with Offshore Teams," in *Hawaii International Conference on System Sciences*, 2009, pp. 1–8.

[8] M. Kajko-Mattsson, G. Azizyan, and M. K. Magarian, "Classes of Distributed Agile Development Problems," in *Agile Conference*, 2010, pp. 51–58.

[9] J. Sauer, "Agile Practices in Offshore Outsourcing - An Analysis of Published Experiences," *Proceedings of the 29th Information Systems Research*, pp. 1–12, 2006.

[10] P. Xu, "Coordination In Large Agile Projects," *Review of Business*, vol. 13, no. 4, pp. 29–44, 2009.

[11] J. Patton, "Secrets to Automated Acceptance Tests." [Online]. Available: www.stickyminds.com/s.asp?F=S13798_COL_2. Accessed: May 2013.

[12] K. Beck, M. Beedle, A. V. Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "Manifesto for Agile Software Development," *The Agile Alliance*, 2001. [Online]. Available: http://agilemanifesto.org/. Accessed: May 2013.

[13] E. Hossain, P. L. Bannerman, and D. R. Jeffery, "Scrum Practices in Global Software Development : A Research Framework," *PROFES 2011*, pp. 88–102, 2011.

[14] "Rally." [Online]. Available: http://www.rallydev.com/. Accessed: May 2013.

[15] "Mingle." [Online]. Available: http://www.thoughtworks-studios.com/mingle-agile-project-management. Accessed: May 2013.

[16] “VersionOne.” [Online]. Available: www.versionone.com. Accessed: May 2013.

[17] “TargetProcess.” [Online]. Available: www.targetprocess.com. Accessed: May 2013.

[18] B. Boehm and R. Turner, *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley, 2003, p. 266.

[19] B. W. Boehm, “Software Engineering,” *IEEE Transactions on Computers*, vol. 25, no. 12, pp. 1226–1241, 1976.

[20] B. Boehm, “Guidelines for verifying and validating software requirements and design specifications,” *Proc European Conf Applied Information Technology*, 1979.

[21] M. Rueher, “Structured rapid prototyping -- an evolutionary approach to software development,” *Information and Software Technology*, vol. 32, no. 4, p. 318, 1990.

[22] B. W. Boehm, “A spiral model of software development and enhancement,” *Computer*, vol. 21, no. 5, pp. 61–72, 1988.

[23] P. Kruchten, *The Rational Unified Process: An Introduction*. Addison-Wesley, 2003, p. 298.

[24] M. Huo, J. Verner, L. Zhu, and M. A. Babar, “Software Quality and Agile Methods,” *Computer*, pp. 1–6, 2004.

[25] I. Sommerville, *Software engineering (9th edition)*. Addison Wesley, 2010.

[26] L. Williams and A. Cockburn, “Agile software development: it’s about feedback and change,” *IEEE Computer*, vol. 36, no. 6, pp. 39–43, 2003.

[27] A. MacCormack, “Product-Development Practices That Work: How Internet Companies Build Software,” *MIT Sloan Management Review*, vol. 42, no. 2, pp. 75–84, 2001.

[28] S. Kharytonov, “Waterfall, RUP and Agile: Which is Right for You,” 2009. [Online]. Available: http://www.executivebrief.com/software-development/waterfall-rup-agile/. Accessed: May 2013.

[29] M. Fowler, “The New Methodology,” *Wuhan University Journal of Natural Sciences*, vol. 6, no. 1–2, pp. 12–24, 2001.

[30] O. Salo, “Enabling Software Process Improvement in Agile Software Development Teams and Organisations,” University of Oulu, 2006.

[31] A. Sillitti, M. Ceschi, B. Russo, and G. Succi, “Managing Uncertainty in Requirements: A Survey in Documentation-Driven and Agile Companies,” *11th IEEE International Software Metrics Symposium METRICS05*, no. Metrics, pp. 17–17, 2005.

[32] J. Highsmith and A. Cockburn, “Agile software development: the business of innovation,” *Computer*, vol. 34, no. 9, pp. 120–127, 2001.

[33] K. Beck, *Extreme Programming Explained*. Addison-Wesley, 1999, p. 224.

[34] K. Schwaber, “Scrum development process,” in *OOPSLA’95 Workshop on Business Object Design and Implementation*, 1995, pp. 10–19.

[35] A. Cockburn, *Surviving object-oriented projects: A manager’s guide*. Addison Wesley, 1998, p. 250.

[36] J. Stapleton, *DSDM: Dynamic Systems Development Method: The Method in Practice*. Addison Wesley, 1997, pp. 1–163.

[37] J. A. Highsmith, “Adaptive Software Development,” *Adaptive software development*, pp. 173–179, 1999.

[38] A. Hunt and D. Thomas, *The Pragmatic Programmer*. Addison-Wesley, 2000.

[39] P. Coad, E. LeFebvre, and J. D. Luca, *Java modeling in color with UML: Enterprise components and process*. Prentice Hall, 1999.

[40] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta, “Agile software development methods,” *Vtt Publications*, vol. 478, no. 3, pp. 167–168, 2002.

[41] J. Highsmith, “History: The Agile Manifesto,” 2001. [Online]. Available: http://agilemanifesto.org/history.html. Accessed: May 2013.

[42] K. Beck, *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2004, p. 224.

[43] K. Beck and M. Fowler, *Planning Extreme Programming*. Addison-Wesley, 2001, p. xvii, 139 p.

[44] “JUnit.” [Online]. Available: www.junit.org/. Accessed: May 2013.

[45] “NUnit.” [Online]. Available: www.nunit.org/. Accessed: May 2013.

[46] “A Metric Leading to Agility,” *XProgramming*, 2004. Accessed: May 2013.

[47] M. Fowler, “Continuous Integration,” *Integration The Vlsi Journal*, 2006. [Online]. Available: http://martinfowler.com/articles/continuousIntegration.html. Accessed: May 2013.

[48] A. Cockburn, *Agile Software Development: The Cooperative Game (Second Edition)*. Addison-Wesley Professional, 2006.

[49] J. Koskela, “Software configuration management in agile methods,” *Vtt Publications*, vol. VTT Public, no. 514, p. 54, 2003.

[50] R. Jeffries, A. Anderson, and C. Hendrickson, *Extreme Programming Installed*. Addison-Wesley, 2001, p. 288.

[51] M. Lippert, S. Roock, and H. Wolf, *eXtreme programming in action: practical experiences from real world projects*. J. Wiley, 2002.

[52] M. C. Paulk, “Extreme programming from a CMM perspective,” *IEEE Software*, vol. 18, no. 6, pp. 19–26, 2001.

[53] “Summary of the subworkshop on extreme programming,” *Nordic Journal of Computing*, vol. 9, no. 3, 2002.

[54] VersionOne, “5th Annual State of Agile Development Survey Final summary report,” 2010.

[55] “Agile Practices and Principles Survey Results,” 2008. [Online]. Available:
http://www.ambysoft.com/surveys/practicesPrinciples2008.html.
Accessed: May 2013.

[56] PMI, *A guide to the project management body of knowledge (PMBOK® guide)*, vol. 40, no. 2. Project Management Institute, 2008.

[57] K. Schwaber, “Agile project management with Scrum,” *Redmond Microsoft Press*, p. 163, 2004.

[58] M. national I. Initiative, “MSC Malaysia Research & Development.” [Online]. Available:
http://www.mscmalaysia.my/codenavia/portals/msc/images/img/business/

grow_your_business/msc_status_funding/rnd_grant_scheme/MGS_MsMeiYuet.pdf. Accessed: May 2013.

[59] P. Haikonen, "Distributed Agile software development and the requirements for Information Technology," Helsinki University of Technology, MSc Dissertation, 2009.

[60] T. Schümmer and J. Schümmer, "Support for Distributed Teams in eXtreme Programming," *Information Systems Journal*, pp. 355–377, 2001.

[61] I. Lehtonen, "Communication Challenges in Agile Global Software Development," in *University of Helsinki, Report*, 2009.

[62] J. D. Herbsleb and R. E. Grinter, "Splitting the organization and integrating the code: Conway's law revisited," in *Proceedings of the 21st international conference on Software engineering*, 1999, vol. Los Angeles, no. 5, pp. 85–95.

[63] T. J. Allen, *Managing the Flow of Technology*. MIT Press, 1977, p. 320.

[64] J. D. Herbsleb and A. Mockus, "An Empirical Study of Speed and Communication in Globally-Distributed Software Development," *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 1–14, 2003.

[65] B. Sengupta, S. Chandra, and V. Sinha, "A research agenda for distributed software development," *Proceeding of the 28th international conference on Software engineering ICSE 06*, vol. 2006, no. 3, p. 731, 2006.

[66] R. J. Ocker, "The relationship between interaction, group development, and outcome: a study of virtual communication," in *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, 2001.

[67] M. Paasivaara, S. Durasiewicz, and C. Lassenius, "Using Scrum in Distributed Agile Development: A Multiple Case Study," in *Fourth IEEE*

International Conference on Global Software Engineering, 2009, pp. 195–204.

[68] E. Therrien, “Overcoming the Challenges of Building a Distributed Agile Organization,” in *Agile Conference*, 2008, pp. 368–372.

[69] W. Williams and M. Stout, “Colossal, Scattered, and Chaotic (Planning with a Large Distributed Team),” in *Agile Conference*, 2008, pp. 356–361.

[70] N. Ducheneaut and V. Bellotti, “E-mail as habitat: an exploration of embedded personal information management,” *interactions Magazine*, vol. 8, no. 5, pp. 30–38, 2001.

[71] E. Bradner, W. A. Kellogg, and T. Erickson, “The adoption and use of BABBLE: A field study of chat in the workplace,” in *Proceedings of the Sixth European conference on Computer supported cooperative work*, 1999, pp. 139–158.

[72] A. Sarma, “A Survey of Collaborative Tools in Software Development,” in *University of California Irvine, ISR Technical Report*, 2005.

[73] L. Layman, L. Williams, D. Damian, and H. Bures, “Essential communication practices for Extreme Programming in a global software development team,” *Information and Software Technology*, vol. 48, no. 9, pp. 781–794, 2006.

[74] M. Korkala, P. Abrahamsson, and P. Kyllonen, “A Case Study on the Impact of Customer Communication on Defects in Agile Software Development.,” in *Agile 2006*, 2006, pp. 76–88.

[75] M. Vax and S. Michaud, “Distributed Agile: Growing a Practice Together,” in *Agile 2008 Conference*, 2008, pp. 310–314.

[76] M. Cottmeyer, “The Good and Bad of Agile Offshore Development,” *Agile 2008 Conference*, pp. 362–367, 2008.

[77] S. H. Rayhan and N. Haque, “Incremental Adoption of Scrum for Successful Delivery of an IT Project in a Remote Setup,” *Agile 2008 Conference*, pp. 351–355, 2008.

[78] A. Cockburn, *Crystal-Clear a Human-Powered Methodology for Small Teams*. Pearson Education, Inc., p. 312, 2005.

[79] A. Danait, “Agile offshore techniques - a case study,” *Agile Development Conference ADC05*, pp. 214–217, 2005.

[80] X. Wang, F. Maurer, R. Morgan, and J. Oliveira, “Tools for Supporting Distributed Agile Project Planning,” *J Phys Conf Ser*, vol. 41, pp. 1–18, 2006.

[81] Google, “Google Spreadsheets.” [Online]. Available: http://www.google.com/google-d-s/spreadsheets/. Accessed: May 2013.

[82] M. Dubakov and P. Stevens, “Agile Tools: The Good, the Bad and the Ugly.” *Report*, TargetProcess, Inc, 2008.

[83] T. Chau and F. Maurer, “A case study of wiki-based experience repository at a medium-sized software company,” *KCAP 05 Proceedings of the 3rd international conference on Knowledge capture*, p. 185, 2005.

[84] Mircrosoft, “MS Project.” [Online]. Available: http://www.microsoft.com/project/en-us/project-management.aspx. Accessed: May 2013.

[85] G. Azizyan, M. K. Magarian, and M. Kajko-Matsson, “Survey of Agile Tool Usage and Needs,” *2011 AGILE Conference*, pp. 29–38, 2011.

[86] P. Behrens, Agile Project Management (APM) Tooling Survey Results, Trail Ridge Consulting,” 2006.

[87] V. Heikkilä, “Tool Support for Development Management in Agile Methods,” Helisinki University of Technology, 2008.

[88] “ScrumWorks.” [Online]. Available: http://www.collab.net/products/scrumworks/. Accessed: May 2013.

[89] “ExtremePlanner.” [Online]. Available: www.extremeplanner.com. Accessed: May 2013.

[90] “XPlanner.” [Online]. Available: http://xplanner.codehaus.org/. Accessed: May 2013.

[91] “Pivotal Tracker.” [Online]. Available: www.pivotaltracker.com. Accessed: May 2013.

[92] “Scrum VSTS.” [Online]. Available: www.scrumforteamsystem.co.uk. Accessed: May 2013.

[93] “Agilefant.” [Online]. Available: www.agilefant.org. Accessed: May 2013.

[94] “IceScrum.” [Online]. Available: www.icescrum.org. Accessed: May 2013.

[95] “Planbox.” [Online]. Available: www.planbox.com. Accessed: May 2013.

[96] “XP StoryStudio.” [Online]. Available: www.xpstorystudio.com/. Accessed: May 2013.

[97] “XPWeb.” [Online]. Available: xpweb.sourceforge.net/. Accessed: May 2013.

[98] “AgileWrap.” [Online]. Available: www.agilewrap.com/. Accessed: May 2013.

[99] "ScrumDesk." [Online]. Available: www.scrumdesk.com/. Accessed: May 2013.

[100] "SpiraTeam." [Online]. Available: www.inflectra.com/spirateam/. Accessed: May 2013.

[101] "Leankit." [Online]. Available: leankitkanban.com/. Accessed: May 2013.

[102] "DevSuite." [Online]. Available: www.techexcel.com/devsuite/. Accessed: May 2013.

[103] "TinyPM." [Online]. Available: www.tinypm.com/. Accessed: May 2013.

[104] "Planigle." [Online]. Available: planigle.com/. Accessed: May 2013.

[105] "Acunote." [Online]. Available: www.acunote.com/. Accessed: May 2013.

[106] "On Time." [Online]. Available: www.axosoft.com/. Accessed: May 2013.

[107] "AgileZen." [Online]. Available: www.agilezen.com/. Accessed: May 2013.

[108] "ScrumPad." [Online]. Available: www.code71.com/. Accessed: May 2013.

[109] "eXPlainPMT." [Online]. Available: https://github.com/explainpmt/explainpmt. Accessed: May 2013.

[110] "AgileBuddy." [Online]. Available: www.agilebuddy.com/. Accessed: May 2013.

[111] "Daily Scrum." [Online]. Available: http://daily-scrum.com/. Accessed: May 2013.

[112] "Express." [Online]. Available: agileexpress.sourceforge.net/. Accessed: May 2013.

[113] "Agile Tracking." [Online]. Available: https://sites.google.com/site/agiletrackingtool/home. Accessed: May 2013.

[114] D. J. Anderson, "Using Cumulative Flow Diagrams." The Coad Letter – Agile Management, *Technical Report*, 2004.

[115] U. Asklund, L. Bendix, and T. Ekman, "Software Configuration Management Practices for eXtreme Programming Teams," in *Proceedings of the 11th Nordic Workshop on Programming and Software Development Tools and Techniques*, 2004, pp. 1–16.

[116] B. Appleton et al., "Lean Traceability: A smattering of strategies and solutions," *CM Journal*, 2007.

[117] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: a change impact analysis tool for java programs," *ACM Sigplan Notices*, vol. 39, no. 10, pp. 664–665, 2004.

[118] H. Mintzberg, *The Structuring of Organizations*. Prentice-Hall, 1979, p. 512.

[119] S. Alyahya, "Investigation of using system dynamics to understand coordination in software projects," *MSc Dissertation,* Cardiff University, 2006.

[120] W. Morris, *The American Heritage Dictionary of the English Language*, Boston: Houghton Mifflin, 2000.

[121] T. W. Malone, "What is Coordination Theory ?," in *Coordination Theory Workshop.*, 1988, vol. 88, no. SSM WP # 2051–88, pp. 357–370.

[122] T. W. Malone and K. Crowston, “What is coordination theory and how can it help design cooperative work systems?,” *Proceedings of the 1990 ACM conference on Computersupported cooperative work CSCW 90*, October, pp. 357–370, 1990.

[123] A. D. Chandler, *Strategy and structure: chapters in the history of the industrial enterprise*. M.I.T. Press, 1962.

[124] J. D. Thompson, *Organizations in Action: Social Science Bases of Administrative Theory*, vol. 48, no. 3. McGraw-Hill, 1967, p. 192.

[125] N. S. Foundation, “A report by the NSF-IRIS review panel for research on coordination theory and technology,” Washington, D.C., 1989.

[126] B. Curtis, “Modeling coordination from field experiments,” in *Organizational Computing Coordination and Collaboration Theories and Technologies for ComputerSupported Work*, 1989.

[127] B. Singh. Interconnected roles (IR): A coordination model. *Technical Report,* CT-084-92, MCC, 1992.

[128] T. W. Malone and K. Crowston, “The interdisciplinary Study of Coordination,” *ACM Computing Surveys*, vol. 26, no. 1, pp. 87–119, 1994.

[129] K. Crowston, J. Rubleske, and J. Howison, “Coordination Theory: A Ten-Year Retrospective,” in *Human-Computer Interaction and Management Information Systems: Foundations*, M.E. Sharpe Inc, 2004.

[130] T. W. Malone, K. Crowston, J. Lee, B. Pentland, G. Wyner, J. Quimby, C. S. Osborn, A. Bernstein, M. Klein, E. O. Donnell, C. Dellarocas, G. Herman, and F. Investments, “Tools a for Inventing of Organizations : Toward Handbook Organizational Processes Michigan Massachusetts,” *Management*, vol. 45, no. 3, pp. 425–443, 1999.

[131] "Team Foundation Server (TFS)." [Online]. Available: http://msdn.microsoft.com/en-us/vstudio/ff637362. Accessed: May 2013.

[132] R. Kass and I. Stadnyk, "Intelligent Assistance for the Communication of Information in Large Organizations," in *Proceedings of 8th Conference on Artificial Intelligence for Application*, 1992, pp. 171–178.

[133] H-T. Chou and W. Kim, "A Unifying Framework for Version Control in a CAD Environment," in *Proceedings of the 12th International Conference on Very Large Data Bases*, 1986, pp. 336–344.

[134] W. K. Ivins, W. A. Gray, and J. C. Miles, "A process-based approach to managing changes in a system to support engineering product design," in *Proc of the Engineering Design Conference*, 2002.

[135] R. H. Katz, M. Anwarrudin, and E. Chang, *A Version Server for Computer-Aided Design Data*. IEEE Press, 1986, pp. 27–33.

[136] Open_University, "Models and modelling." [Online]. Available: http://openlearn.open.ac.uk/mod/oucontent/view.php?id=397581§ion=3.1. Accessed: May 2013.

[137] D. E. Avison and G. Fitzgerald, *Information Systems Development: Methodologies, Techniques and Tools*. McGraw-Hill, 2006, p. 608.

[138] M. S. and F. R. Bennett S, *Object Oriented System Analysis and Design: using UML*. McGraw-Hill, 1999.

[139] P. Stevens, *Using UML: Software Engineering with Objects and Components*. Addison Wesley, 2005.

[140] A. M. Langer, *Analysis and Design of Information Systems.* Springer, New York., 2010.

[141] “Object Management Group.” [Online]. Available: http://www.omg.org/. Accessed: May 2013.

[142] “Fitnesse.” [Online]. Available: http://fitnesse.org/. Accessed: May 2013.

[143] “Selenium.” [Online]. Available: http://seleniumhq.org/. Accessed: May 2013.

[144] R. Jeffries, “Re: Problems with Acceptance Testing.” [Online]. Available: http://xprogramming.com/xpmag/problems-with-acceptance-testing/. Accessed: May 2013.

[145] J. Shore, “The Problems With Acceptance Testing.” [Online]. Available: http://jamesshore.com/Blog/The-Problems-With-Acceptance-Testing.html. Accessed: May 2013.

[146] “GO.” [Online]. Available: http://www.thoughtworks-studios.com/go-agile-release-management. Accessed: May 2013.

[147] “Pulse.” [Online]. Available: http://zutubi.com/products/pulse/. Accessed: May 2013.

[148] “Teamcity.” [Online]. Available: http://www.jetbrains.com/teamcity/. Accessed: May 2013.

[149] Gauntlet, “Gauntlet.” [Online]. Available: http://techpubs.borland.com/silk_gauntlet/gauntlet/2007_10/en/Gauntlet_10.pdf. Accessed: May 2013.

[150] D. Poon, “A Self Funding Agile Transformation,” in *Agile Conference*, 2006, pp. 342–350.

[151] “Plasticscm.” [Online]. Available: http://www.plasticscm.com/. Accessed: May 2013.

[152] "Reducing the Impact of Broken Builds." [Online]. Available: http://zutubi.com/products/pulse/articles/brokenbuilds/. Accessed: May 2013.

[153] "Subversion." [Online]. Available: subversion.tigris.org. Accessed: May 2013.

[154] "Clearcase." [Online]. Available: http://www-01.ibm.com/software/awdtools/clearcase/. Accessed: May 2013.

[155] "Accurev." [Online]. Available: http://www.accurev.com/. Accessed: May 2013.

[156] P. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk*. Addison-Wesley Professional, 2007, p. 336.

[157] C. A. Ellis, S. J. Gibbs, and G. Rein, "Groupware: some issues and experiences," *Communications of the ACM*, vol. 34, no. 1, pp. 38–58, 1991.

[158] J. Grudin, "Groupware and social dynamics: eight challenges for developers," *Communications of the ACM*, vol. 37, no. 1, pp. 92–105, 1994.

[159] W. Orlikowski, "Learning from notes: Organizational issues in groupware implementation," *The Information Society*, vol. 9, no. 3, pp. 237–250, 1993.

[160] D. Pinelle and C. Gutwin, "A review of groupware evaluations," in *Proceedings IEEE 9th International Workshops on Enabling Technologies Infrastructure for Collaborative Enterprises WET ICE 2000*, 2000, pp. 86–91.

[161] R. M. D. Araujo, F. M. Santoro, and M. R. S. Borges, The CSCW lab ontology for groupware evaluation, *The 8th International Conference on Computer Supported Cooperative Work in Design*, 2004.

[162] J. M. Carroll, “Making Use : Scenarios and Scenario-Based Design,” *Design*, pp. 1998–1998, 2000.

[163] S. R. Haynes, S. Purao, and A. L. Skattebo, “Scenario-Based Methods for Evaluating Collaborative Systems,” *Computer Supported Cooperative Work*, vol. 18, no. 4, pp. 331–356, 2009.

[164] O. Stiemerling and A. B. Cremers, “The use of cooperation scenarios in the design and evaluation of a CSCW system,” *IEEE Transactions on Software Engineering*, vol. 24, no. 12, pp. 1171–1181, 1998.

[165] Smart Bear Software, “Peer Code Review : An Agile Process,” in *the proceedings of the Agile Development Practices conference*, 2009.

[166] L. Crispin and T. House, *Testing Extreme Programming*. Addison-Wesley Professional, 2002.

[167] “NetBeans IDE”, [Online]. Available: http://www.netbeans.org. Accessed: May 2013.

[168] “MySQL”, [Online]. Available: http://www.mysql.com/. Accessed: May 2013.

[169] T. Zimmermann, “Changes and bugs — Mining and predicting development activities,” *2009 IEEE International Conference on Software Maintenance*, pp. 443–446, 2009.

[170] A. E. Hassan and R. C. Holt, “Predicting change propagation in software systems,” in *20th IEEE International Conference on Software Maintenance 2004 Proceedings*, 2004, pp. 284–293.

[171] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, “Mining version histories to guide software changes,” in *IEEE Transactions on Software Engineering*, 2005, vol. 31, no. 6, pp. 429–445.

[172] S. V. Shrivastava and H. Date, “Distributed Agile Software Development: A Review,” *Journal of Computer Science*, vol. 1, no. 1, pp. 10–17, 2010.

[173] L. Bendix and C. Pendleton, “Configuration Management–Mother’s little helper for Global Agile Projects?,” in *International Conference on Global Software Engineering-Workshop*, 2012.

[174] K. Keefe and M. Dick, "Using Extreme Programming in a capstone project." In *Proceedings of the Sixth Australasian Conference on Computing Education-Volume 30*, pp. 151-160. Australian Computer Society, Inc., 2004.

[175] M. Yap, "Follow the sun: distributed extreme programming development." In *Proceedings of the Agile Conference (2005)*, pp. 218-224. IEEE, 2005.

[176] D. Wells, "Working Software", [Online] http://www.agile-process.org/working.html, Accessed: May 2013.

[177] J. Sutherland, A. Viktorov, J. Blount, and N. Puntikov, "Distributed Scrum: Agile Project Management with Outsourced Development Teams" In in *HICSS'40, Hawaii International Conference on Software Systems*, Big Island, Hawaii, 2007.