Managing Versioning Activities to Support Tracking Progress of Distributed Agile Teams

Sultan Alyahya, Wendy K. Ivins, W. A. Gray
School of Computer Science & Informatics, Cardiff University,
Queen's Buildings, 5 The Parade, Cardiff, CF24 3AA, U.K

ABSTRACT

Development Progress in agile methods is based on the amount of "working software" completed by team members. Changes to the source code might be introduced that affect the working software. Team members face difficulties in understanding and sharing changes that affect development progress especially in distributed projects. They may not recognise that there is an effect, or may not know who is affected by a change. In addition, changes are not perceived by the current tracking systems and hence if these changes affect development progress, they will not be discovered. This may lead to weak awareness of development progress and extra defects and delays. In this paper, we attempt to support tracking distributed agile projects by identifying and co-ordinating the impact of versioning activities on development progress, thereby ensuring that progress information is more consistent with the current software state. This will provide distributed agile teams with improved transparency of the actual progress.

General Terms

Software Engineering, Agile Development.

Keywords

Distributed Agile Development, Versioning System, Co-ordination, Progress Tracking.

1. INTRODUCTION

Software agile projects are broken down into short iterations where a set of requirements are implemented in each iteration. Requirements in agile methods like extreme programming [1] are defined in terms of user stories (i.e. feature), each of which represents a unit of functionality of the system. Each user story is split into one or more tasks that may be undertaken by different developers.

Measuring progress in agile development is based on the completion of the software artefacts (source code) required to implement user stories. One of the principles introduced by the agile manifesto [2] is that: *working software is the primary measure of progress.*

The source code artefacts produced by a task/story have to achieve certain technical criteria before the task/story state can be deemed to be completed. A task is complete only if source code artefacts associated with it are unit-tested. Additionally, a user story is complete when source code artefacts associated with it are integrated and acceptance-tested. Any change to source code artefacts may lead to a change in tasks/stories progress.

If the project is co-located, team members can be made aware of the impact of versioning activities on development progress through their interactions with other team members. The ad-hoc co-ordination is likely to facilitate partial sharing of the progress information among team members. When project is distributed, team members find it harder to maintain an awareness of development progress. Numerous distributed agile projects reported difficulties in tracking development progress (e.g. [3,4]).

While there are many progress tracking systems developed (e.g. Rally [5], Mingle [6], TargetProcess [7], VersionOne [8]), these have no provision for identifying and co-ordinating the effect of versioning activities on progress tracking systems in agile projects. Tracking systems are isolated from versioning systems and are fed by the perceptions of team members about the progress of their tasks.

Ulf et al. [9] 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. Brad et al. [10] 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 change impact.

In our research, we attempt to support tracking distributed agile projects by identifying and co-ordinating the impact of versioning activities on development progress, thereby ensuring that progress information is more consistent with the current software state.

The remainder of the paper is organised as follows: section 2 will investigate the current methods of tracking progress of agile projects. Section 3 will highlight the role of source code versioning in agile methods and how it affects development progress. Section 4 and 5 will introduce a new approach to tracking progress system and will describe the design aspects of it. Then, system validation and general discussion are

provided in section 6 before we conclude the paper in section 7.

2. CURRENT METHODS TO TRACKING AGILE DEVELOPMENT PROGRESS

The current progress tracking systems provide detailed information about iterations' tasks and stories. They show the breakdown of a user story into tasks. Task size is determined by the number of hours estimated for the task. As work progresses, team members move tasks/stories from one state (i.e. unstarted, in progress, complete) to another.

The main limitation of the current tracking systems is that these systems are static and rely completely on team members to realise change in progress. Changes caused by versioning activities (e.g. modifying source code) are not perceived by the current tracking systems and hence if these changes affect development progress, they will not be discovered.

Rally, TargetProcess and VersionOne have started providing integrations with some versioning systems. These tools allow developers to post updates to tasks and source code without taking precious time to log their activity in both systems. However, these integrations are insufficient to realise the impact of the versioning activities on development progress.

Furthermore, in co-located teams, face-to-face communication and daily stand-up meetings enable team members to share changes that may affect development progress. In distributed teams, meetings can be held by video-conferencing tools, though these are often held less frequently than stand-up meetings, and teams may rely more on asynchronous communications (e.g. email).

This manual approach has many limitations. The impact of a change may not be fully recognised in team members' perceptions because of the difficulty in understanding the impact of change brought about by the work of one team member on the work of others. Team members may not recognise that there is an effect, or may not know who is affected by a change. In addition, meetings may communicate change but it is up to each developer to realise which changes affect their own work.

3. SOURCE CODE VERSIONING

3.1 Versioning in Agile Methods

Agility is about creating and responding to change [11]. For this reason, most agile methods recommend using versioning systems to automate the change process. According to Cockburn [12], 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 [13]. Since rapid development and quick changes may lead to mistakes in development, it is important that earlier versions of artefacts are accessible.

Ron Jeffries et al [14] pointed out that there should be as few restrictions as possible in a versioning system, 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 [15], who found that optimistic concurrency control is a superior locking mechanism in agile methods like XP.

3.2 The Impact on Development Progress

Since agile methods consider working software as the main measure of progress, 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, updating a source code version that belongs to a completed story means that the story is no longer deemed to be complete.

There is, therefore, a need to manage change to the source code so that it becomes clear how it influences development progress. Managing source code change should include providing co-ordination activities such as checking progress constraints, identifying potential source of progress change, reflecting progress change in the tracking system, finding and notifying team members affected by potential progress change [16]. An example of a check-out process is provided in the next section to clarify the need for managing change impact on development progress.

3.3 Example: Check-out Process

A typical check-out process will require the following co-ordination activities:

- *Ensure corresponding task and story are still active:* developers can only work on active task/story.

- *Change progress state of the corresponding task/story:* if the task or the story is inactive, it must be changed to active. This needs to be explicitly shown on the tracking system so that the whole team will have an awareness of the actual progress of the project.

- *Find and notify affected team members:* affected team members (i.e. story owner/tester) may be in different sites. It is important to look for affected team members and notify them in order to resolve problems as early as possible.

The co-ordination required for the check-out process has been described in the following diagram (Figure 1):

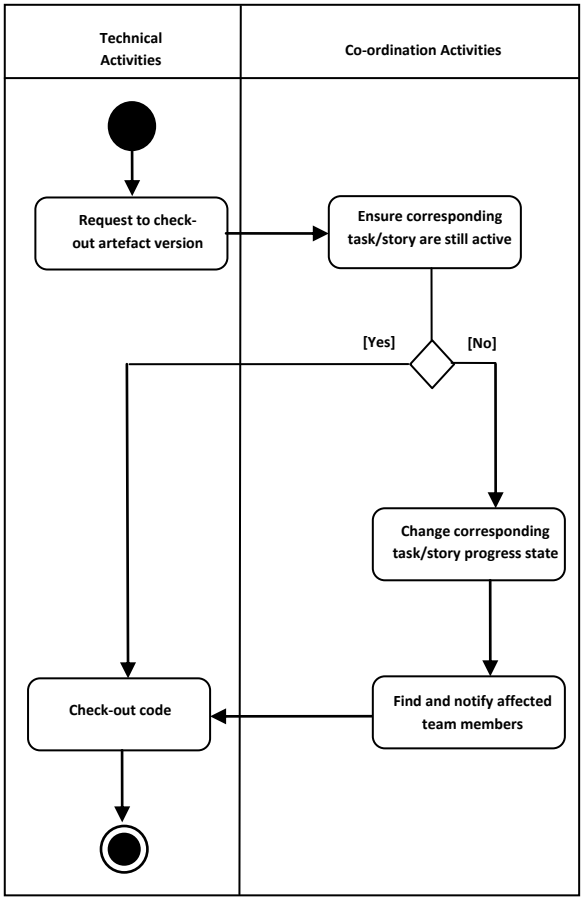

Fig 1: Co-ordination required for the check-out process.

Current progress tracking systems do not support identifying and co-ordinating impact of versioning activities on development and hence co-ordination activities are performed in a manual manner. Due to limitations of the manual-based approach, as analysed in section 2, a new approach is required.

4. A NEW APPROACH TO DESIGN A PROGRESS TRACKING SYSTEM

Because development progress in agile development is directly based on the maturity of the source code artefacts, it was realised that versioning activities should be the heart of developing a progress tracking system.

Current versioning systems provide technical mechanisms to store and control source code artefacts but it provides no support for identifying how changes can affect development progress and provide no support for co-ordinating versioning activities affecting development progress.

4.1 Integrating Versioning Data and Progress Tracking Data

Tasks/stories should not be tracked separately from the source code artefacts that determine their functionalities. There should be a consistency between tracking data and a team member's actual work (e.g. if a task is complete this must indicate that software produced by the task is unit-tested). It is required to track the influence of the versioning activities on the development progress in order to know which task/story is affected due to a change.

4.2 Unit-Testing Impact on Artefact Evolution

There is a need to differentiate between two types of artefacts: development artefacts (source code) and unit-testing artefacts. This distinction is mainly because they are different entities that have explicit relationship between them.

The existence of unit-testing artefact is mandatory for each development artefact. The status of unit-testing artefact needs to be 'pass' before checking-in the development artefact.

4.3 Acceptance-Testing Impact on Artefact Evolution

Acceptance tests are high level tests of user stories and are used to ensure that software developed for the stories meet customer requirements.

Acceptance testing is related to the releasing process. Source code artefacts can not be released if one or more of acceptance tests that the artefact belong to has failed. It is required to link each artefact not only with the relevant tasks and unit tests but also with the relevant stories and acceptance tests.

4.4 Continuous Integration (CI) Impact on Artefact Evolution

The idea of CI is to run the build and integration tests regularly, over a short period of time [17]. Usually it is done in an asynchronous manner by tools such as Go [18]. Code should be integrated before completing the corresponding story.

The integration result has direct impact on development progress because it is a condition to complete stories. Most of the current agile tracking systems give attention to the integration result but do not show its impact on project progress. They do not show how the integration result influences source code artefact evolution. An integration 'pass' result should contribute to making progress to affected stories. In addition, the 'failed' result should not affect those stories that do not have new versions entered in the build.

4.5 Progress Change Notifications

Team members need to be notified about changes that may affect progress of tasks/stories that they work on. Current tracking systems do not notify for changes resulting from code change and the current versioning systems that provide notification mechanisms do not consider its relevance with development progress.

Not every versioning activity has a direct effect on other team members. Therefore, developers need an effective progress change notification mechanism that targets only affected team

members as a key requirement of tracking development progress.

5. DESIGN

As development progress in agile methods is determined by the maturity of the source code artefacts, the tracking system should tightly integrate task/story data with the versioning data. The proposed tracking system (Figure 2) keeps track of both of them.

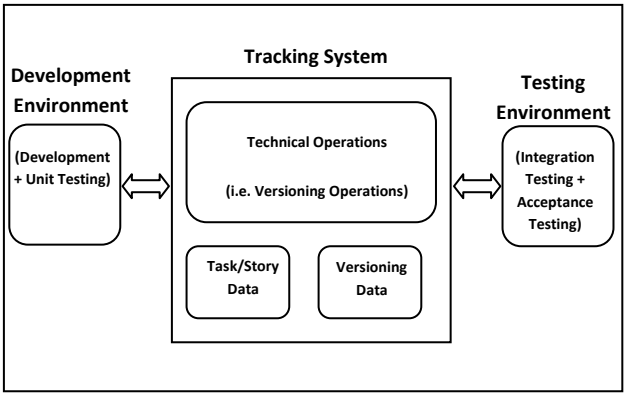

Fig 2: A conceptual design of tracking system for agile development.

5.1 Versioning Model

Version states are used to indicate the maturity of different versions of source code artefacts. Version state is taken into account when determining task/story 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:

● 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 'Ready-for-Integration' stage.

● Integrated Version (IV): the artefact version is unit-tested and has passed the build.

● Releasable Version (RV): The user stories that artefact version provides functionality for, have passed AT and ready for releasing.

The use of hierarchical structure in a versioning system is not new. It has been widely proposed for versioning systems built to support managing change in software design and engineering design (e.g. [19, 20, 21]).

One of the advantages of providing version states is that developers become aware of the state of the version they work on. They can choose which version they want to use; either the latest version which probably unit tested only or choose an integrated version which is more stable and reliable.

5.2 Versioning Operations

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

Table 1: Extended versioning Operations.

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	A version can be released to the customer.
Delete an artefact	An artefact is deleted.

The UML Statechart Diagram in Figure 3 shows how the new versioning operations can change an artefact state.

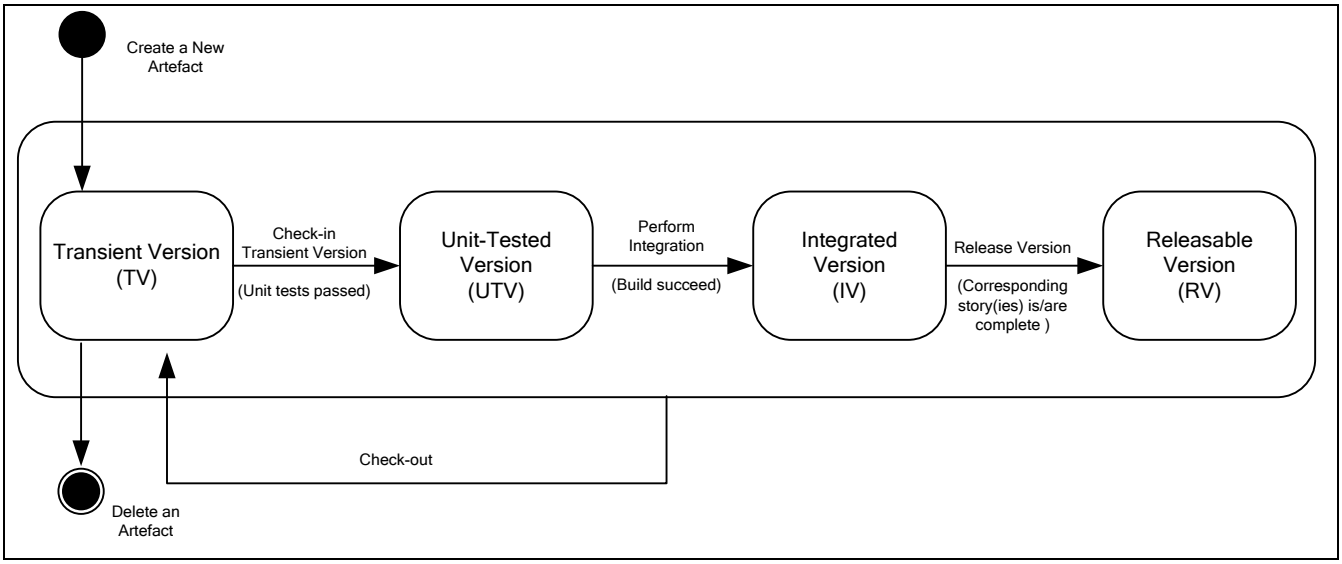

Fig 3: Source code version states.

Source code artefacts related to a completed story may be versioned and need to be re-integrated and re-tested. As a result, in addition to the normal user story states: 'Not started', 'Active' and 'Complete', we have added two more states: 'Waiting for integration' and 'Waiting for Acceptance Testing'. These states can provide more accurate progress information. They reflect the effect of the versioning activities on the story's progress.

5.3 Data Model

In an agile project, there is a large number of dependencies among tasks, stories, releases, unit tests, acceptance tests and integration tests. The dependencies among the various entities in the tracking system need to be carefully represented in a data model. The UML Class Diagram in Figure 4 shows the relationships among the main entities in the system.

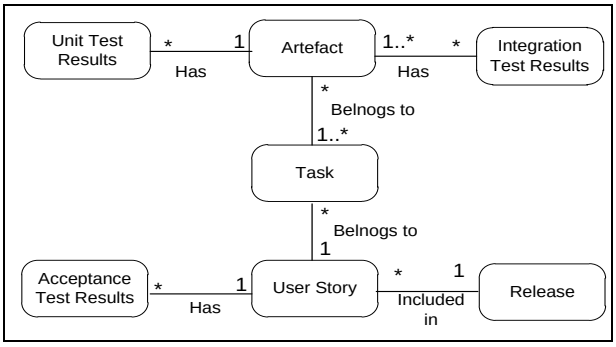

Fig 4: UML Class Diagram for the main entities in the tracking system.

5.4 Modeling Versioning Activities

A set of process models is developed to cover the versioning operations shown in Table 1. These models provide one possible way to model the versioning operations. Different agile projects may have different requirements based on their working practices. Therefore, the proposed models can be adapted.

Each process model illustrates how each versioning activity affects development progress. A visual representation of the co-ordination activities required has been provided. This includes explicit support for checking progress constraints, finding and notifying affected team members by progress change, identifying potential source of progress change and reflecting progress change in the tracking system.

A UML Activity Diagrams were chosen as techniques to model the versioning activities. They are able to provide transparent processes that explicitly show the co-ordination required to support tracking progress. They are also capable to provide behavioral model to clearly represents both sequential and concurrent activities. Figure 5 shows the 'check-out' process model.

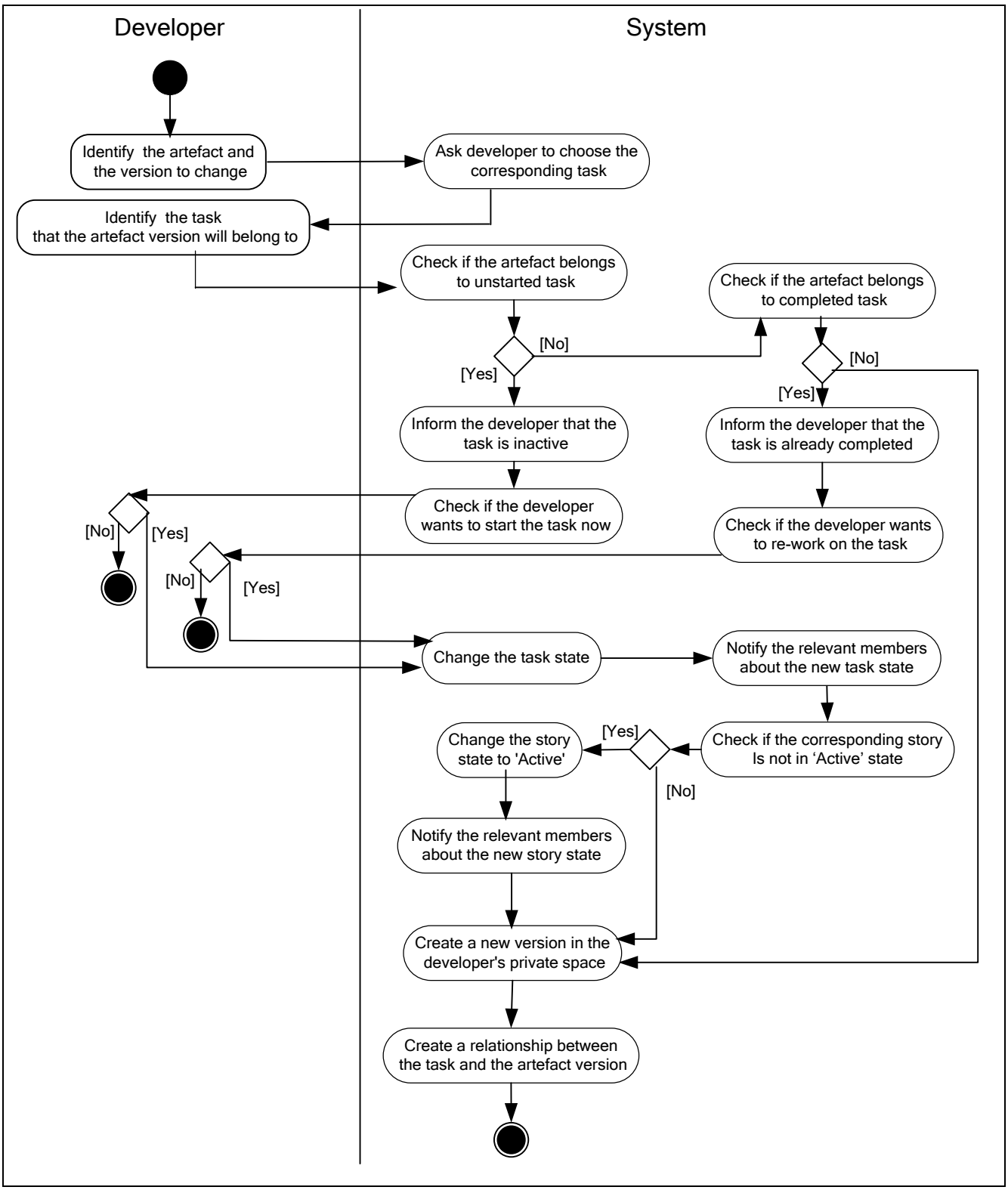

Fig 5: The 'Check-out' process model.

6. VALIDATION AND DISCUSSION

As a proof of concept, a Java application, including implementation of several scenarios, has been used to validate the tracking system (Figure 6). The implementation shows to us that the new approach can be made. The data model and the SQL queries were able to identify the different dependencies. In addition, the process models were able to provide the necessary co-ordination (e.g. notifications).

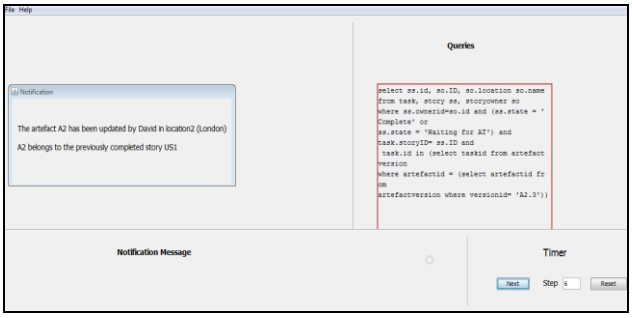

Fig 6: Implementation of the progress tracking system.

In order to evaluate our approach to developing a progress tracking system, we made a number of analytic comparisons between the classical approach to use the versioning activities and our approach. As an example, the check-out process is used in this paper to highlight the differences between the two approaches. Based on the co-ordination requirements for the check-out process (presented in section 3.3), the two approaches are compared and contrasted (Table 2).

It is clear from the comparison that classical versioning approach relies completely on ad-hoc co-ordination to manage the work. Given the limitations of the ad-hoc co-ordination (section 2), a lack of awareness about the actual development progress may occur.

The new approach provides better visibility of the actual work completed by developers' tasks. It immediately identifies potential change in progress resulting from the check-out process. By doing so, the new approach will help team members identify sources of potential defects that may cause project delay at earlier time.

Our approach links each task/story with the functionalities (source code) performed to fulfill its requirements. Changes that directly affect story functionalities are recognised due to the linkage. However, changes affecting progress could be as a result of implicit relationship between the functionalities of one story and another. This is not addressed in our approach. Change impact analysis techniques (e.g. [22,23]) is a hot topic in the literature of software engineering and it is outside the scope of this research.

Table 2: A comparison between the classical approach to use the versioning activities and the new approach.

Co-ordination Activity	Classical Versioning Approach	New Approach
Ensure corresponding task and story are still active	● No formal checking. Developer usually does not consider the task/state progress state.	● System checks if the versioning operation affects progress. Progress change is identified once the versioning operation is used.
Change progress state of corresponding task/story	● Progress change is not reflected in the tracking system.	● System reflects progress change in the tracking system.
Find and notify affected team members	● It is done in an ad-hoc manner. Developer will need to figure out who must be contacted and then he/she will need to share information with them informally	● It is automated by the system once versioning operation is used.

7. CONCLUSION

We proposed in this paper a new approach to developing a progress tracking system for distributed agile development. The approach supports identifying and co-ordinating the effects of the various versioning activities on development progress.

The approach helps minimising the inconsistency between the progress information and the actual software produced. Because the approach helps in reducing the cycle time of discovering the defects, The velocity of the team is more likely to improve. In addition, progress metrics (e.g. RTF [24]) can be more safely used by project management as these metrics will be based on more realistic progress information.

This research supports using agile practices for distributed projects because it contributes in solving awareness problems associated with managing changes in these projects.

8. ACKNOWLEDGMENT

The first author is sponsored by King Saud University, Kingdom of Saudi Arabia.

9. REFERENCES

[1] Beck, K., Extreme Programming Explained: Embrace Change, Addison-Wesley, 2004.

[2] Agile Alliance, *Manifesto for Agile Software Development*, URL: http://www.agilemanifesto.org/principles.html, Cited 15 January 2012.

[3] Sauer, J., Agile Practices in Offshore Outsourcing - An Analysis of Published Experiences. *Proceedings of the 29th Information Systems Research*, 2006.

[4] Peng Xu - Coordination In Large Agile Projects *Review of Business Information Systems* (RBIS), 2011.

[5] Rally, URL: http://www.rallydev.com/, Cited 15 January 2012.

[6] Mingle, http://studios.thoughtworks.com/mingle-agile-project-management, Cited 15 January 2012..

[7] TargetProcess, URL: http://www.targetprocess.com/, 15 January 2012..

[8] VersionOne, URL: http://www.versionone.com/, 15 January 2012.

[9] Asklund, U., Bendix, L. & Ekman, T., 2004. Software Configuration Management Practices for eXtreme Programming Teams. *Management*, p.1-16. Cockburn, A., Highsmith, J. Agile Software Development: The Business of Innovation. Computer, 2001, Vol. 34, No. 9, pp. 120–122.

[10] Appleton, B., Cowham, R., and Berczuk, S., Lean Traceability: A smattering of strategies and solutions, *CM Journal*, 2007.

[11] Cockburn, A., Highsmith, J., Agile Software Development: The Business of Innovation, *IEEE Computer*, 2001, Vol. 34, No. 9, pp. 120–122.

[12] Cockburn, A. Agile Software Development. Boston: Addison-Wesley, 2002.

[13] Koskela, J., Software Configuration Management in Agile Methods, *VTT Publication*, Finland, 2003.

[14] Jeffries, R., Anderson, A., Hendrickson, C. Extreme Programming Installed. NJ: Addison-Wesley, 2001.

[15] Lippert, M., Roock, S., Wolf, H. Extreme Programming in Action: Practical Experiences from Real World Projects. England: John Wiley & Sons, 2002.

[16] Alyahya, S., Ivins, WK., Gray, WA., Co-ordination Support for Managing Progress of Distributed Agile Projects, *IEEE International Conference on, pp. 31-34, 2011 IEEE Sixth International Conference on Global Software Engineering Workshop*, 2011.

[17] Fowler, M., Continuous Integration. *Integration The Vlsi Journal*, 26(1), p.1-6. 2006, Available at: http://martinfowler.com/articles/continuousIntegration.html

[18] Go, URL: http://thoughtworks-studios.com/go-agile-release-management. Cited 15 January 2012.

[19] Katz, R.H., Chang E. and Anwarrudia M. "A Version server for Computer-Aided Design Data. In *Proceedings of ACM/IEEE 23rd Design Automation Conference*, Las Vegas, U.S.A, pp 27-33, Jun 1986.

[20] Chou, H.T. and W. Kim.: "A Unifying Framework for Version Control in a CAD Environment" In *Proceedings of the 12th International Conference on Very Large Databases*, Kyoto, Japan, pp 336-344, Aug 1986.

[21] Ivins W K, Gray W A, Miles J C, A process-based approach to managing changes in a system to support engineering product design, *Proc of the Engng Design Conf* , (2002) 469-478 ISBN 1 86058 372 5.

[22] Zimmermann, T., *Changes and Bugs: Mining and Predicting Software Development Activities*, Books on Demand Gmbh, 2009.

[23] Geipel, M., & Schweitzer, F.: Software change dynamics: evidence from 35 java projects, *ESEC/SIGSOFT FSE*, 2009.

[24] Jeffries, R., A Metric Leading to Agility, in XProgramming, 2004.