Automatic Task Update in Agilefant

Marko Klemetti

T-86.5150 Special Assignment on Enterprise Information Systems P

Mar 23, 2010

Table of Contents

1 Introduction

The past few years have been a revolution to the mainstream of software development when even the largest IT companies have realized the benefits of agile development. Responsibility and code ownership has been handed back to the developers and teams from oppressive management and middle-layer of the companies' organizational structure. Software is developed in short iterations instead of long development steps, and the customer representatives have become a vital part of successful projects.

At the same time the development environments and programming languages have evolved so that the initial learning curve and code overhead has almost disappeared. The developers can concentrate on the program logic instead of the overhead of low-level programming languages like memory management. Code level testing has become a standard for almost all of the programming languages and suddenly the developers can become testers at the same time they are writing new code. The unit testing and iterative development enables better documentation and structuring of the code, further making it possible for teams to develop features instead of owning separate software components.

Since the automatic building and integrating has become popular in software development, it is only natural to enable a flexible way of updating task statuses using automation. Instead of manually verifying that developed features actually work, the team could create a build configuration, which verifies both the already developed features and the new features work as planned.

The purpose of this study is to define different workflow patterns in individual developer's software development practices, and to find the most suitable combination of tools for the development work and automatic feature verification. This work first clarifies the roles and expected behaviors for team members, and then identifies the automatic and manual steps in software development. This work presents four different workflow models for an individual developer, after which the models are discussed and evaluated.

1.1 Research Framework

This study is based on the experiences from different customer organizations that have all successfully adopted agile development practices and have started using continuous integration as a part of their development work.

1.2 Objectives

There are 4 objectives in this study:

1. **How the development workflow enables automatic task update?**
   The objective is to define different development workflows and come up with a model how the workflows could include automatic update of tasks in a backlog management tool.

2. **How the tasks could be automatically updated?**
   The objective is to define the technical requirements for the automatic task updates.

3. **When is the automatic task update feasible?**
   The objective is to define when the automatic task update is feasible and when it should not be implemented as a part of the developer's workflow.

4. **How to implement the automatic task update functionalities in Agilefant?**
   The objective is to define the requirements for developing the automatic task update functionality to the Agilefant tool.

1.3 Scope

The scope of this study is limited to presenting the workflow models using the tools that are currently available, and discussing the possible extensions. The theory behind these methods is presented where it is necessary for the reader to understand the text without additional material.

1.4 Structure of this Work

The content of this work is as follows:

Chapter 1 has presented the introduction to this work.

Chapter 2 briefly discusses a background theory as a bibliographical review.

Chapter 3 presents the development workflow for a single developer using a task board or electronic backlog management tool (Agilefant).

Chapter 4 presents the example application, which is then implemented using four different development workflows.

Chapter 5 evaluates the models presented in chapter 4.

Chapter 6 presents the different options for developing the tool connectors to Agilefant.

Chapter 7 discusses the conclusions of this research work.

Chapter 8 presents the validity of this study and discusses ideas for future research.

Appendix I contains the instructions to implementing the presented workflows in chapter 4.

Appendix II contains the code examples that have been left out of the work content.

2 Background

2.1 Agile Software Development

The agile software development model enables adaptive and inspective software development by breaking the development into small increments. At the end of each increment the results are presented to the stakeholders and further steps are decided. This way the development can adapt to changes quickly and possible long-term risks are minimized. Typically the iterations last from one week to four weeks. The iterative development practice has been found effective by many different agile trailblazers from the Lean Development [1], Extreme Programming [2] and traditional agile Development [3].

The other important feature of agile is teamwork. The development is divided between cross-functional teams, which are completely self-organizing. The team members finish their tasks from design to acceptance testing together, sharing the responsibility of getting the requirements done before the end of iteration [4] and [5].

2.1.1 Scrum

One of the most popular iterative development frameworks that follow agile principles is Scrum. It defines a set of practices and predefined roles that enable the self-organizing and committed work by teams and their members. While the needs, ideas and influences of all stakeholders are taken into account, the team is isolated so that the stakeholders interfere with the team's work [3].

In the scrum process the software development is divided into *sprints*, a fixed length period typically from two to four weeks. The sprint framework is illustrated in Figure 1. During the sprint, the team creates a *working increment* of the software. The working increment is often referred to as *potentially shippable product*, to emphasize the fact that the features should be completely finished [5].

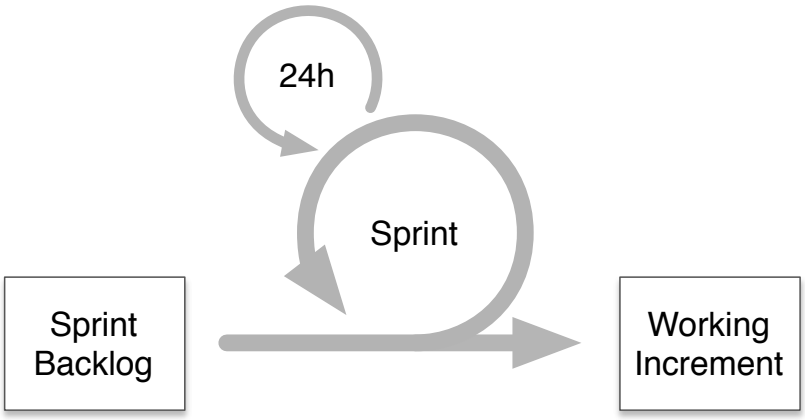


Figure 2-1: The Cycle of one Sprint [6]

The sprint starts with a *sprint planning* session, where the tasks for the next sprint will be selected together with the customer representative. The team is then isolated for the rest of the sprint so that they can make their best effort to change the requirements into a working increment of the software.

During the sprint the team meets each other officially once a day in a meeting called *daily scrum*. In the daily scrum, each of the developers answer to three questions: *what have you done since yesterday*, *what are you planning to do today* and *what obstacles do you have*. The meeting starts at the same time every day, can last maximum of fifteen minutes and only the team members are allowed to speak [3].

2.2 User Stories

User stories define system requirements in one or two sentences using the common language of the user. The representation of a user story is usually a hand-written paper note and the purpose of a user story is to connect the actual users of the system and the developer who implement the features that enable the actual execution of the user story.

In her article Rachel Davies states “user stories present customer requirements rather than document them” [7]. According to Mike Cohn’s description, *user story* describes function that is valuable to user, purchaser, system or software. The users stories consist of three parts [6]:

1. Written story used for planning and as a reminder
2. Conversations about the story that serve to flesh out the details of the story
3. Tests that convey and document details and that can be used to determine when a story is complete

The paper note, i.e. the first step contains the condensed text of the story, while the story is worked out in the second step and recorded in the third [6].

Mike Cohn suggests that the users stories should be formed as sentences in the following form "As a <type of user>, I want <some goal> so that <some reason>" [8]. One option for the *user story* of a thirsty customer presented in the previous chapter is shown below.

```
As a customer
I want to buy a cold beverage using coins
So that I could quench my thirst
```

2.3 Test-driven Development

> "Code without tests is bad code. It doesn't matter how well written it is; it doesn't matter how pretty or object-oriented or well-encapsulated it is. With tests, we can change the behavior of our code quickly and verifiably. Without them, we really don't know if our code is getting better or worse."
>
> (Feathers, 2005)

Test-Driven Development is a practice first presented by Kent Beck (2003). The motivation for developing this practice was based on wanting to create "clean code that works". He presented the practice as a method of driving the development with automated tests [2].

2.3.1 Red – Green – Refactor

These two rules imply that the developer must be able to receive rapid feedback from the running development environment. Kent Beck then derived these two rules into industrially well-known *TDD mantra "*Red – Green – Refactor*"*:

- **Red**: Write a little test that doesn't work, and perhaps doesn't even compile at first.
- **Green**: Make the test pass with as little coding as possible.
- **Refactor**: Eliminate all of the duplication created in merely getting the test to work.

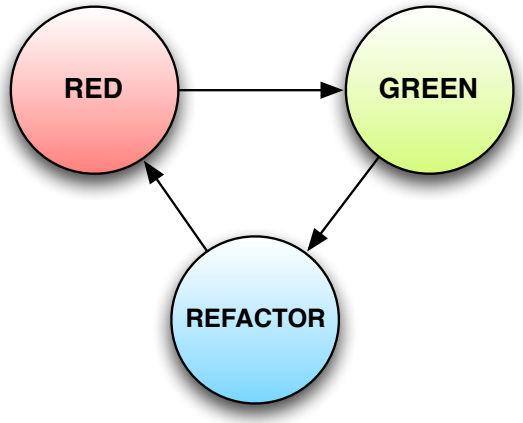


Figure 2-2: Illustration of the Red-Green-Refactor Development Cycle

2.3.2 Test-driven Development Cycle

The test-driven development follows the sequence of development steps, which are continuously repeated:

1. **Add a test**
2. **Run all tests and see the new one fail**
3. **Make a little change**
4. **Run all tests and see them all succeed**
5. **Refactor to remove duplication**

2.4 Definition of Done

Each agile team should have their own set of Definition of Dones [9] for each step of their development process. Definition of Done is a set of rules which explicitly states what needs to happen for an item to be called *Done*. There is usually different set of rules for a task to be called done, and for a feature (or a story) to be done, and the definition of done for a feature (or story) might require that all of the tasks are done according to their Definition of Done.

For a feature or a story the Definition of Done could be for example [10]:

- **All tasks of a story should have at least one automated acceptance test.**
- **The story should have working code supported by unit tests that provide around 60 – 70 percent coverage.**
- **The story should have well defined acceptance criteria.**
- **Code must be completely checked in to the source control system and the build should pass with all the automated tests running.**

2.5 Continuous Integration

Martin Fowler presented the idea of continuous integration already in 2000. He defined continuous integration as a practice where members of the team integrate their code frequently and each integration is verified by an automatic build to detect errors as quickly as possible [11].

The continuous integration Cycle is presented in Figure 2.

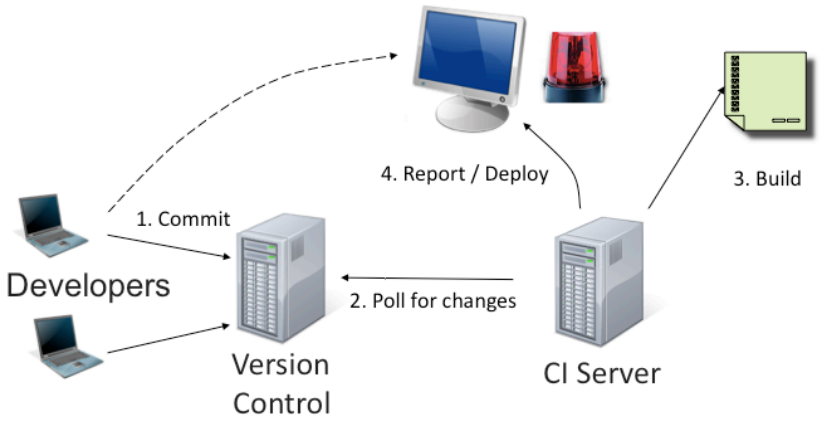


Figure 2-3: The Continuous Integration Cycle

Paul Duvall describes continuous integration by defining the following steps that must be followed [12]:

- All developers run private builds on their own workstations before committing their code to the version control repository to ensure that their changes don't break the integration build.
- Developers commit their code to a version control repository at least once a day.
- Integration builds occur several times a day on a separate build machine.
- 100% of tests must pass for every build.
- A product is generated (e.g., WAR, assembly, executable, etc.) that can be functionally tested.
- Fixing broken builds is of the highest priority.
- Some developers review reports generated by the build, such as coding standards and dependency analysis reports, to seek areas for improvement.

2.5.1 Benefits of Continuous Integration

In the authors experience the best benefits of this practice are:

- **Shorter Feedback Time**
  Through the automated framework every stakeholder of the project can know the latest status of the project. Short feedback also shortens the time between when the defect is introduced and when it's fixed [12].

- **Better Visibility**
  Since the continuous integration reports can be seen online with a browser, everybody can go and see the current status and reports of the project.

- **Remove Repetitive Manual Processes**
  In order to do continuous integration, the build process has to be automated. This way the often manual build processes are automated.

- **Developers have greater confidence**
  When the developers know that there is an automatic process working as a safety net, it is easier to take risks and try out things that would be avoided, often causing a quality loss.

Needless to say that these benefits all cause overall *quality improvement* of both the software and the development.

2.5.2 Hudson in Action

In this work the Hudson tool is used as the continuous integration server, because it's an open source tool and the community is the most active at the time of writing. The Hudson Project Dashboard view is shown in Figure 2-4. The dashboard shows the status and latest information of all projects (content area), status of the currently building projects (lower left) and the configuration options (left menu).

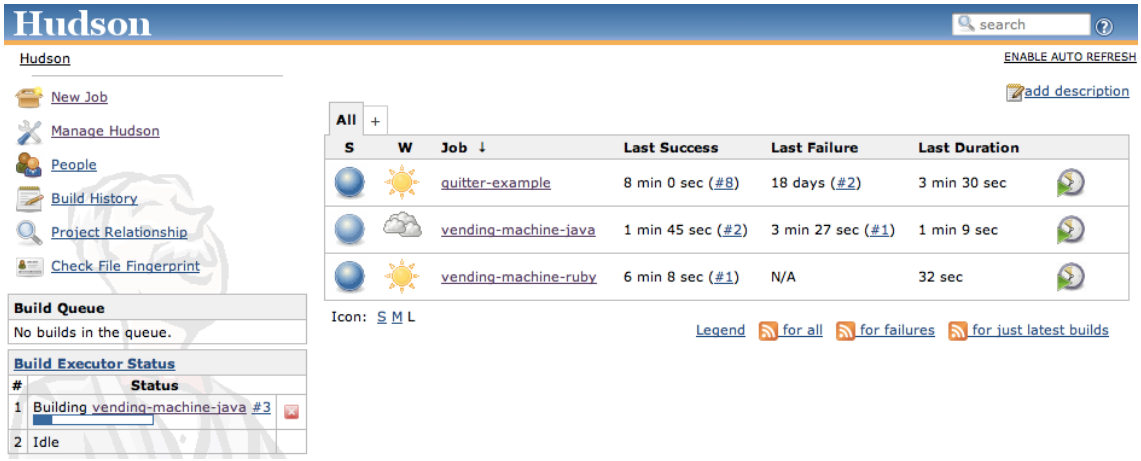


Figure 2-4: The Hudson Dashboard

For a single project the Hudson shows latest results and all the analysis that has been connected to the project. For the *quitter-example* –project the Figure 2-5 shows *latest statistics*, *test result trend* and *test coverage*.

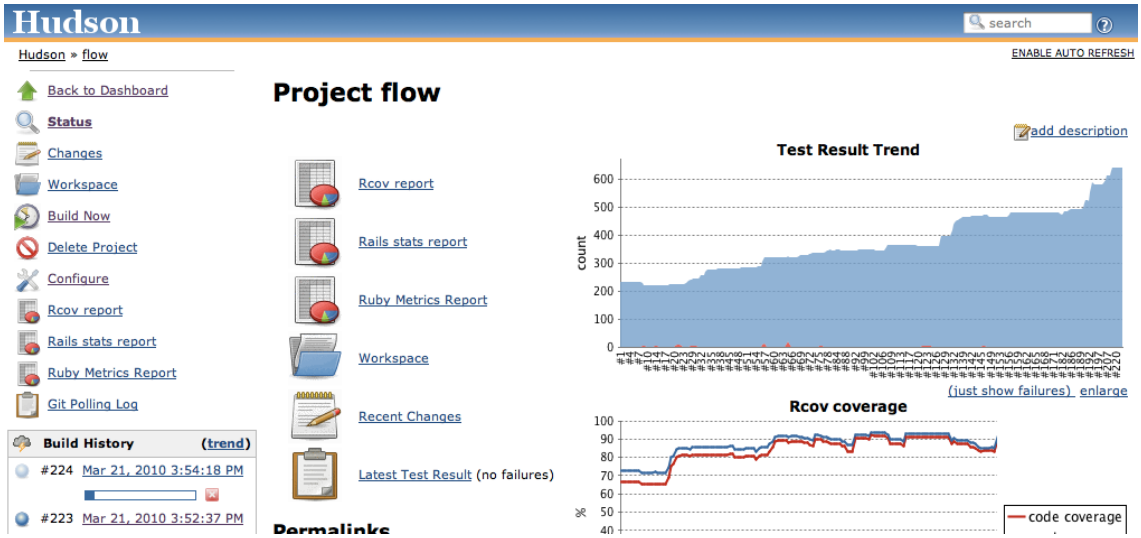


Figure 2-5: Project main view

2.6 Acceptance Testing

The acceptance tests are not the same as unit tests, since their purpose is to make sure that the functionality works from the user's point of view, hence the name. The Acceptance tests should be automatic, even though sometimes the acceptance tests are executed manually.

Often the acceptance tests are developed with another test framework like Robot [34], FIT [35] or Cucumber [36], since the test cases are more extensive and treat the tested system as a black box [13]. They could e.g. do complete operations like adding items to database through a user interface or test a WebService –inteface through a running test server. The acceptance tests take longer time to run than unit tests and they are usually not run as a part of the developer's private build.

The acceptance tests could be developed together with all of the team members, and then enabled in the build at the same time as the task or story is finished. On the other hand the team could have a dedicated tester, who writes the acceptance tests and the developer then enables them as they get the functionality ready for testing. Since the acceptance tests are usually automatic, they should be included as a part of the continuous integration Build.

3 Agile Development Workflow

This chapter describes the developer's regular *agile* or more specifically *Scrum* workflow of a task. The workflow has been observed as two separate processes: a manual process using physical task board with yellow notes and an online process using the Agilefant backlog management tool.

3.1 Workflow using a Scrum Task Board

Traditionally once the Sprint backlog has been set, the tasks are written on paper notes and attached to a scrum task board, which contains all of the tasks from the team's sprint backlog (see Figure 3-1).

The tasks are then developed so that the developers pick a task from *To Do* -column, and move it to *In Progress* -column. Once the task is *verified* and finished according to the team's definition of done, the developer moves it to *Done*. The Developer then picks a new task, moves it to *In Progress* and starts working with the new task. This workflow is illustrated in Figure 3-2.

If the team is estimating the work used, the estimates are usually updated to the notes during the *Daily Scrum*. Once the task is done, the remaining estimates are crossed out.

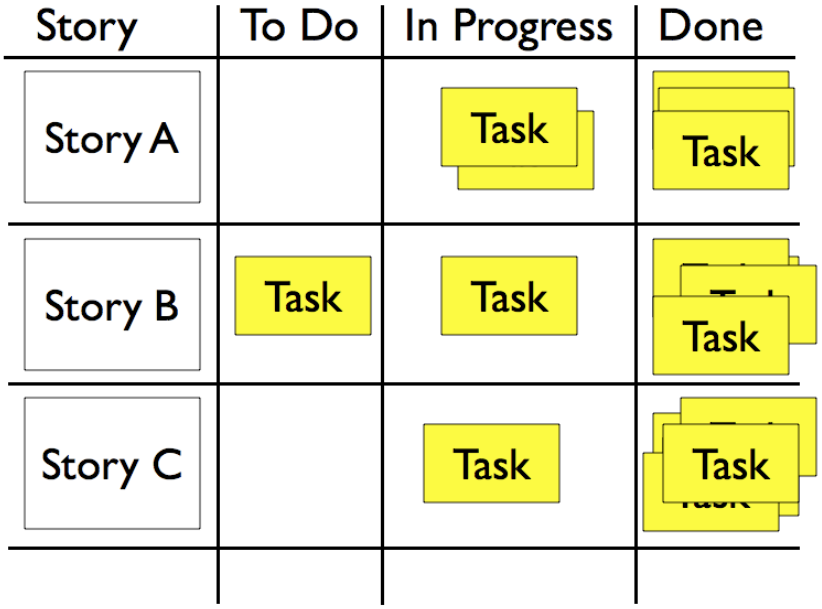


Figure 3-1: Manual Task Board [15]

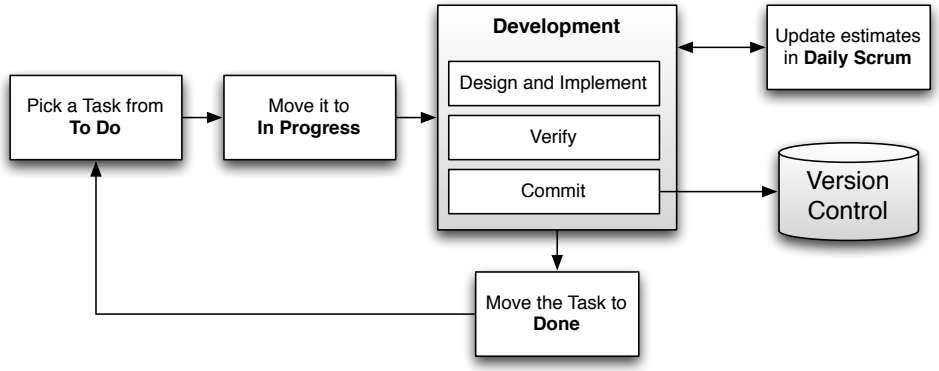


Figure 3-2: The task workflow for a Developer

3.2 Workflow using an Electronic Project Management Tool

The developers can also use an electronic tool as the agile backlog management tool. This work concentrates on Agilefant [16]. The purpose of Agilefant is to integrate the daily work and long-term product and release planning. With Agilefant the teams can manage the backlogs and the developers can manage their tasks, task statuses and effort estimates on daily basis. This enables better visibility into the project status and offers more information for product development planning.

In Agilefant the tasks contain:

- **State**: Not started (similar to To Do), Started, Pending, Blocked, Implemented and Done
- **Responsible**: The team member who is currently responsible (implementing) the task
- **Original Estimate**: The original hourly estimate of the task workload. In manual workflow this is usually marked in the lower right corner of the task note.
- **Estimated Left**: The estimated amount of work left for the task

These task properties enable similar workflow shown in Figure 5.

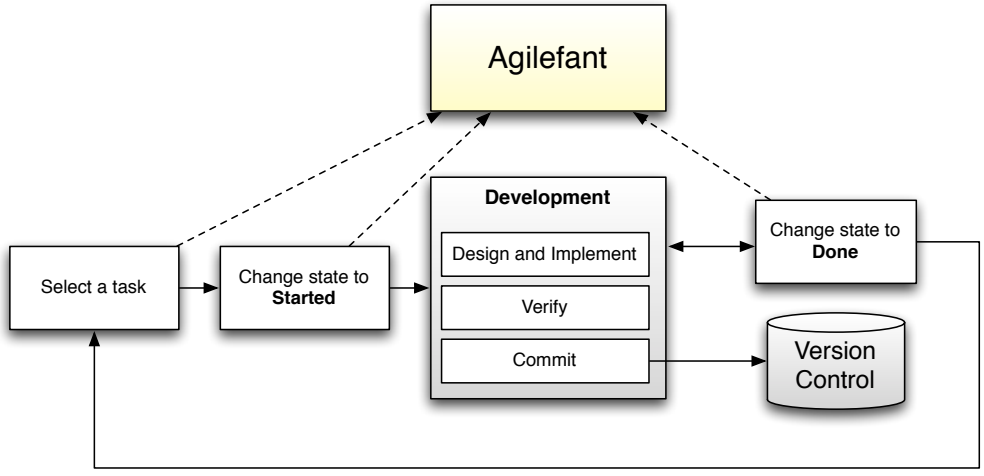


Figure 3-3: Task development using Agilefant

3.2.1 Workflow using an IDE Connector to an Electronic Tool

For some of the electronic agile backlog and defect management tools there are existing connectors to the commonly used IDE's (Integrated Development Environment). For example the Eclipse development IDE contains a connector tool called Mylyn, which can be connected to some of the most popular project and defect management tools like Rally, Mingle, Trac, Jira, Bugzilla, Mantis and Google Code. In this work we will concentrate on the Rally tool, since although not an open source solution, the Rally tool is feature-wise similar to and already has existing connector solutions.

With the combination Eclipse [17] + Mylyn [18] + Rally [19] the development cycle would be the following:

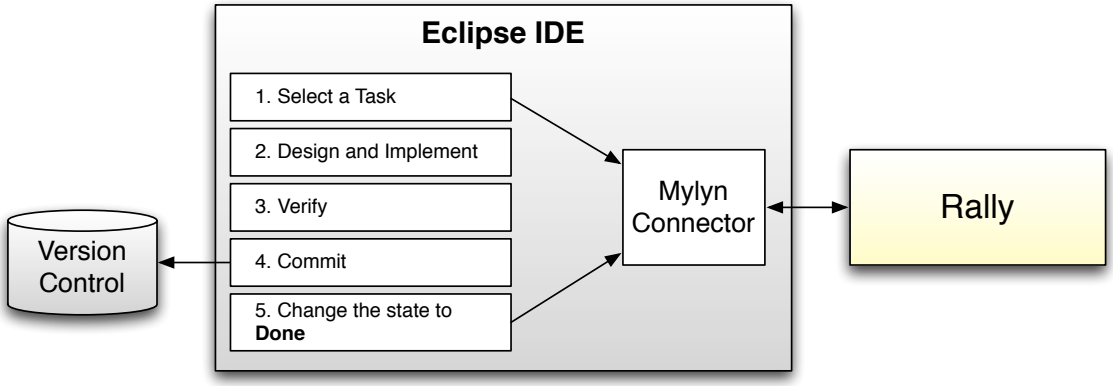


Figure 3-4: Development cycle using IDE and a connector to an online project management tool

When using Mylyn, the developer can easily manage the project statuses in the same place he is developing the features. The task view in Eclipse + Mylyn is illustrated in Figure 3-5. The same tasks in the Rally agile Project Management tool are illustrated in Figure 3-6 and in Agilefant Figure 3-7.

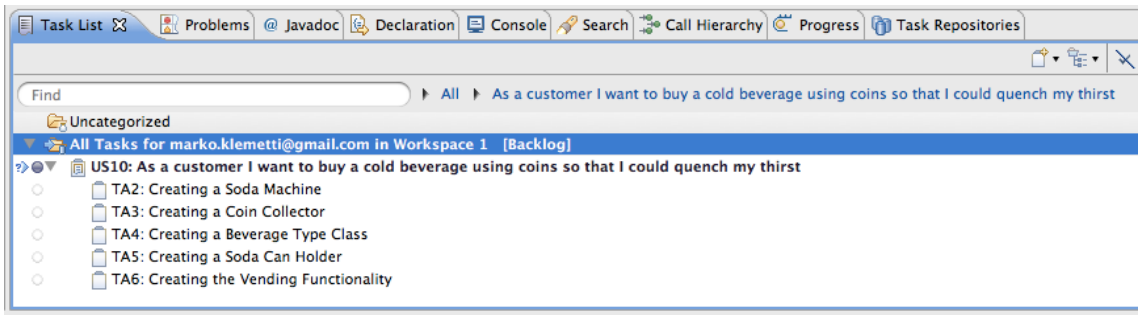


Figure 3-5: Task view in Eclipse + Mylyn

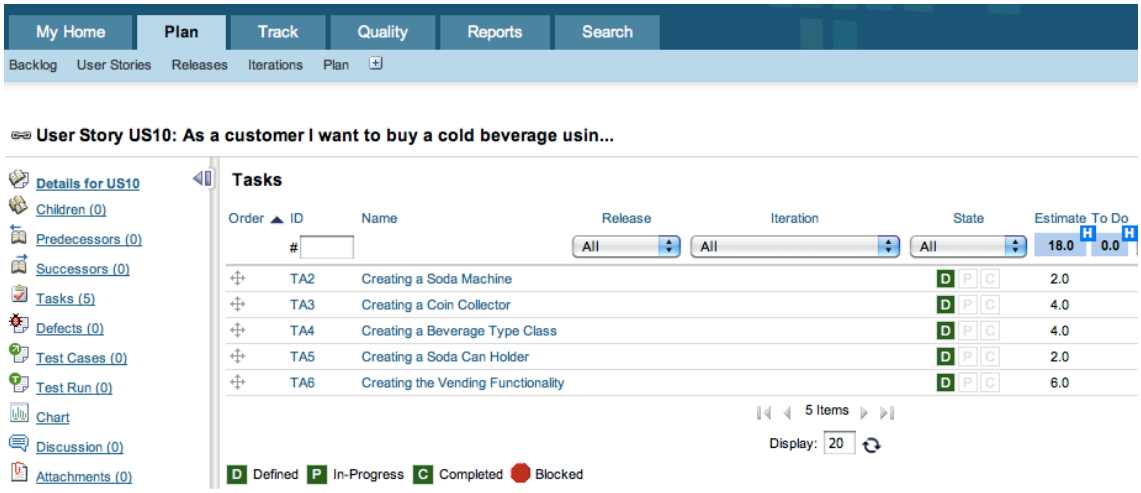


Figure 3-6: Task view in Rally

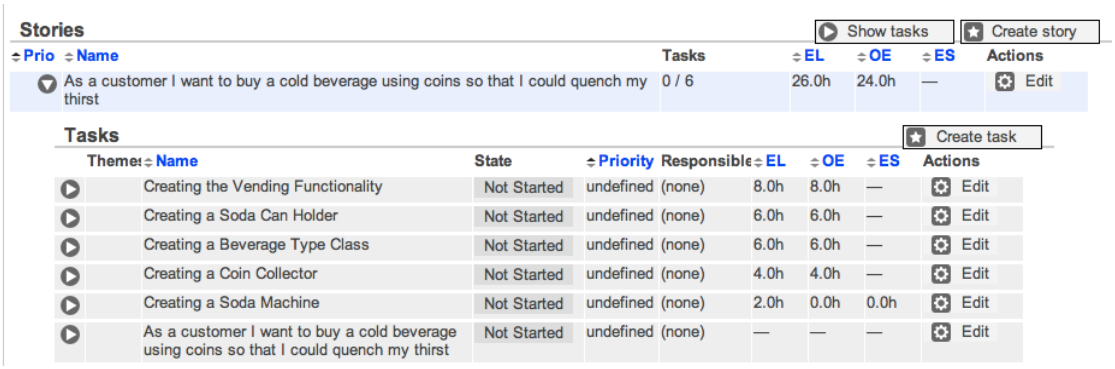


Figure 3-7: Task view in Agilefant

4 Task Updating Models

This chapter discusses four different development workflows for one example feature presented in section 4.1. The workflows are:

1. Completely manual development
2. Development using an IDE and a Connector
3. Development using continuous integration
4. Development using IDE and continuous integration.

To demonstrate the workflow models, the same feature will be implemented with all of the four workflows. For the *manual* and *continuous integration* demonstrations the Ruby programming language is used, and for development using the *IDE* the Java programming language and the Eclipse IDE are used.

In this chapter the Agilefant tool is used for demonstrating the online task management workflow when developing with *manual* or *CI* approaches. For the *IDE* development models the combination *Java*, *Eclipse*, *Mylyn* and *Rally* is used. These tools have been presented in the corresponding section.

4.1 The Example Application

In order to demonstrate the development workflow models, we will develop a simple Soda Vending Machine application. This vending machine will have three essential features: inserting coins, selecting the drink and vending the correct can. These functionalities can be defined with the following user story:

```
As a customer
I want to buy a cold beverage using coins
So that I could quench my thirst
```

This *user story* is split further to *tasks* by the development team. In this example the user story is split into five subtasks:

- Creating a Vending Machine
- Creating a Coin Collector
- Creating a Coda Can
- Creating a Soda Can Holder
- Creating the Vending Functionality

As already stated in the second chapter, the stories and tasks should not be defined too thoroughly on the paper. The most important part of the requirement specification is the knowledge exchange and discussion behind the story and task topics.

4.2 Developing a Task Manually

For the manual development we will use the Ruby programming language and Git version control [20] system provided by Github [21]. The files associated to this section can be found from:
git@github.com:mrako/vending-machine.git

Now as we follow the section 3.1.2 for the manual development, the first thing to do is to log in to Agilefant, select the correct task and mark it as **Started**. This operation is illustrated in Figure 4-1.

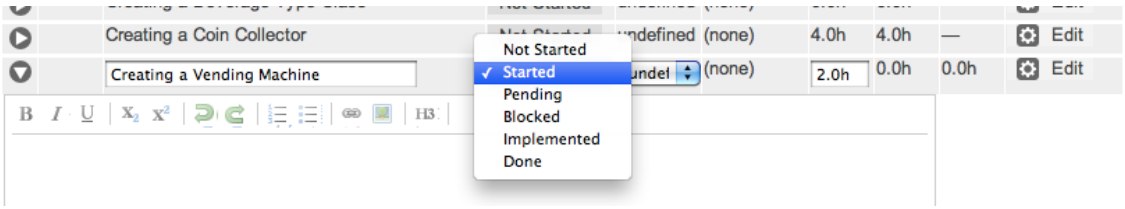


Figure 4-1: Setting a task Started

Then we can proceed to implementing the feature using Test-driven Development. Let's first write a test (spec/unit/vending_machine_spec.rb):

```
require File.dirname(__FILE__) + '/../spec_helper'

require 'vending_machine'

describe VendingMachine do
  before :each do
    @vending_machine = VendingMachine.new
  end

  it "should initially exist" do
    @vending_machine.should_not be_nil
  end
end
```

Then we need to make sure that the test fails:

```
$ rake spec

./spec/unit/vending_machine_spec.rb:3:in `require': no such file to
load -- vending_machine (LoadError)
```

And continue to implement the feature to (vending_machine.rb):

```
class VendingMachine

end
```

Then verify that the test passes:

```
$ rake spec
.

Finished in 0.044637 seconds

1 example, 0 failures
```

At this point there is obviously no need for refactoring (see the Test-driven Development loop in section 2.3.1).

Once we are satisfied with the result and the tests pass, we can commit it to the version control system (github.com repository):

```
$ git commit –am “Created the Vending Machine class”
$ git push origin master
```

And then log back into the Agilefant and mark the task done:

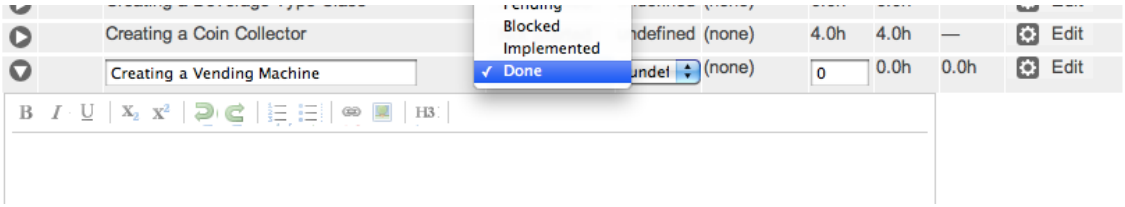


Figure 4-2: Marking a task done in Agilefant

4.3 Developing a Task using Eclipse and Mylyn

For the IDE development we will use Java programming language. The files associated to this section can be found from Google Code [22]:
https://Agilefant-connector-example.googlecode.com/svn/trunk/vending-machine/java

Now we will follow the IDE development workflow in section 3.1.3. Let's first startup Eclipse, select the task from Mylyn, set it to **In-Progress** and click **Submit** (Figure 4-3). The change is immediately synchronized to the Rally server.

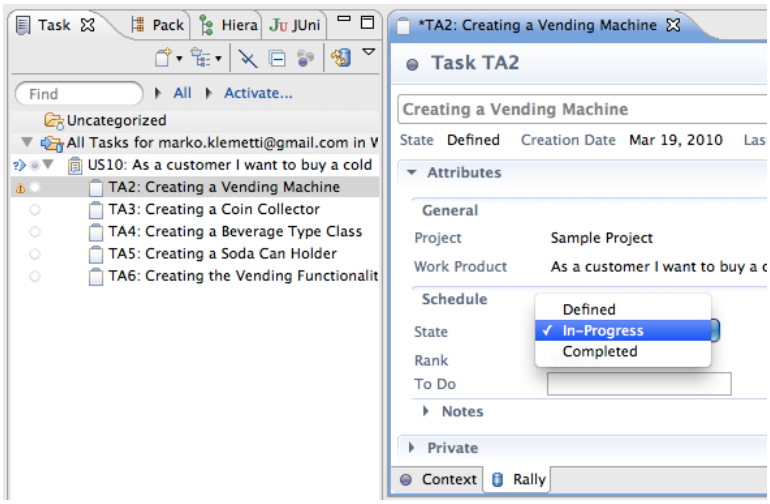


Figure 4-3: Selecting a task and setting it to In-Progress in Mylyn (with Rally connector)

Next we are ready to implement the *Vending Machine* class. Let's start again by writing a test the same way we did in the last section, but this time in Java:

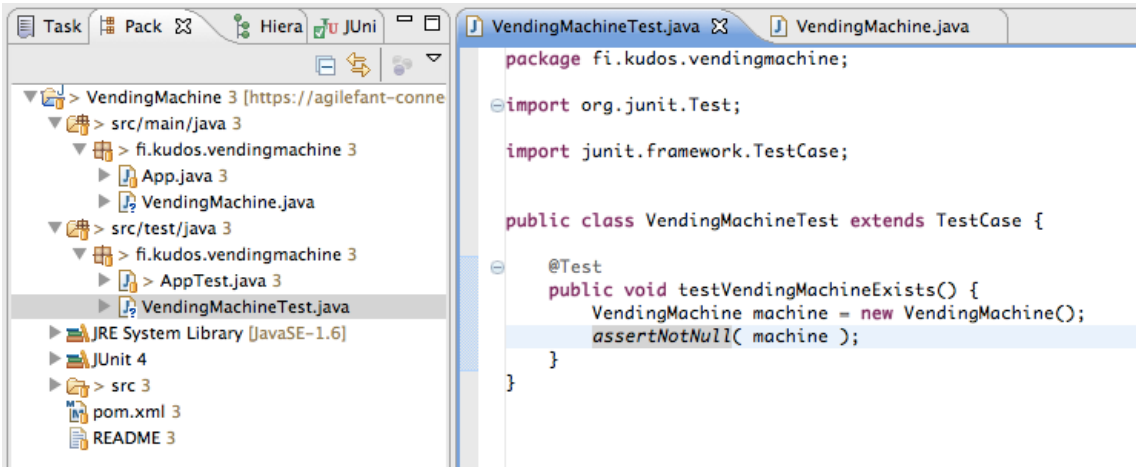


Figure 4-4: The first test case in Java using Eclipse

Usually the IDE forces the developer to create the missing class, before being able to run the tests. After adding it we can run the test case and verify that it passes:

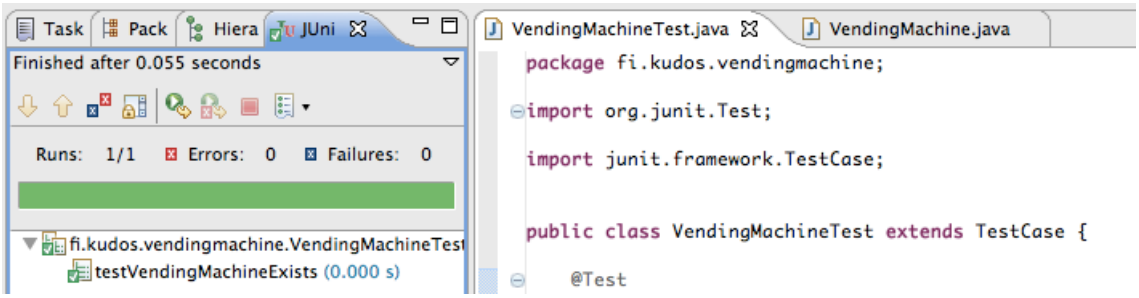
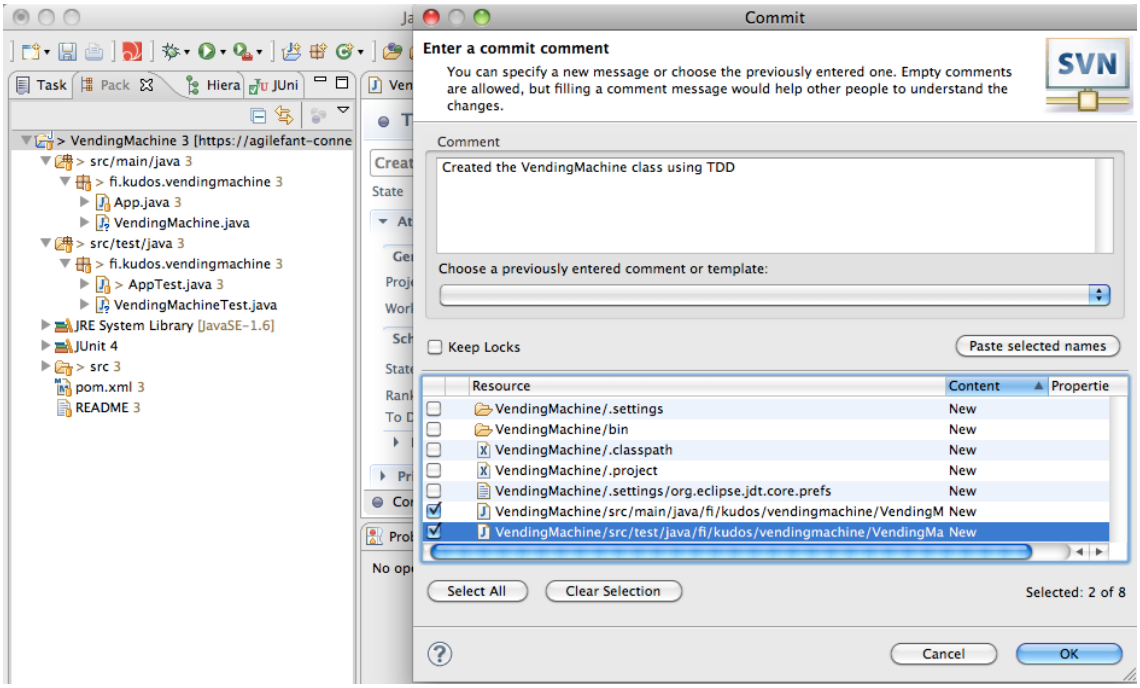


Figure 4-5: Running the jUnit test case with Eclipse

Now that we have successfully implemented and verified the creation of VendingMachine class, we are ready to commit the changes to the version control system (for Java we are using the Google Code Subversion -repository and the Eclipse Subversive -plugin):



Once the changes have been committed, we are can update the task status with Mylyn by going back to the *Tasks* –tab and setting the task ”creating a Vending Machine” to *Completed*:

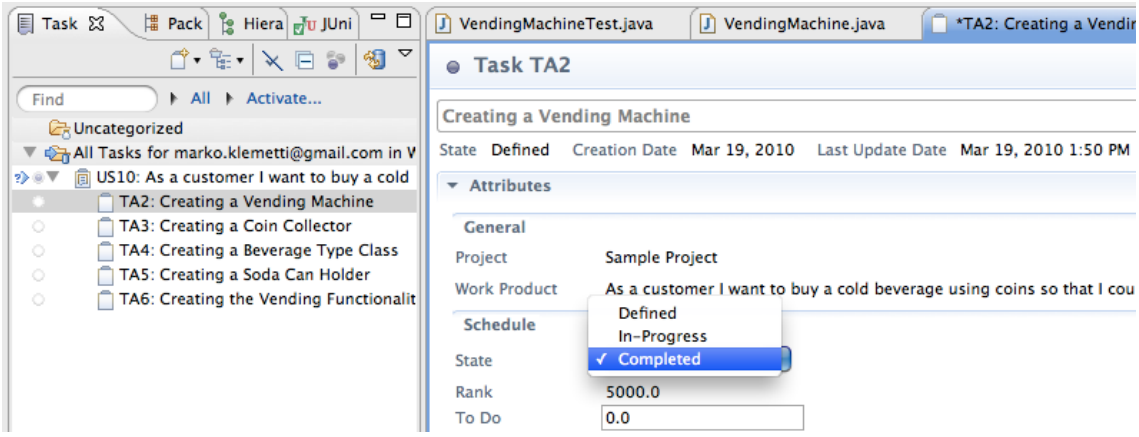


Figure 4-6: Setting a task completed in Eclipse

Additionally we could see from the continuous integration server or from any of the CI radiators like a lavalamp [37] that the build has passed before marking the task done in Mylyn (this is discussed further in section 4.5). There is also a plugin for Eclipse to show the build results automatically in the IDE [38].

The continuous integration build is shown in Figure 4-7 and the task update in Rally is shown in Figure 4-8.

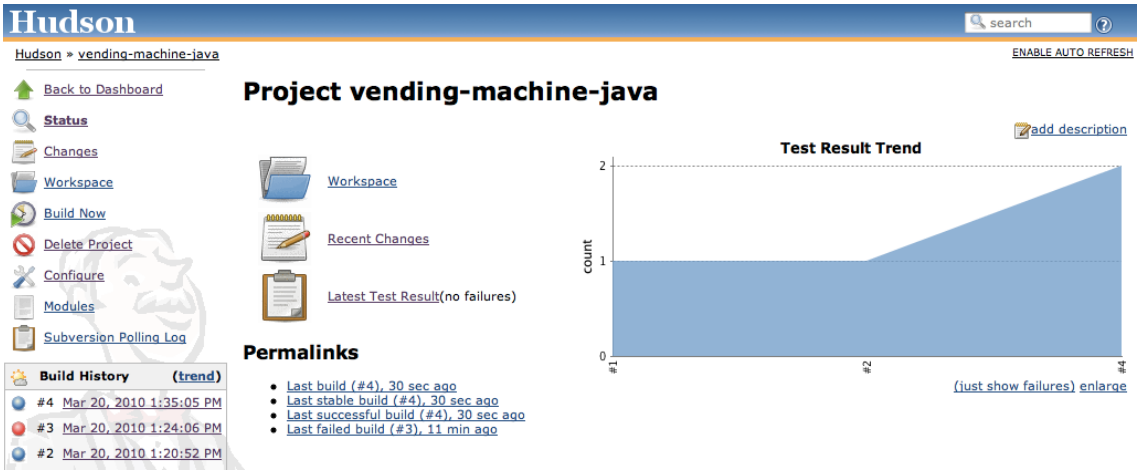


Figure 4-7: continuous integration Build results after the Commit from Eclipse

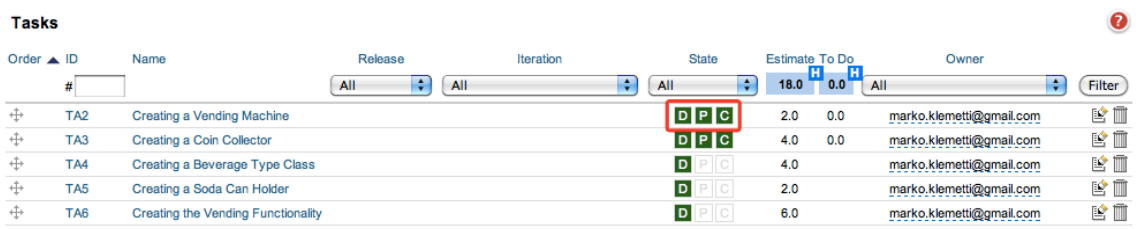


Figure 4-8: Task update in Rally (marked red)

4.4 Developing a Task using continuous integration

The instructions for setting up the continuous integration server can be found from Appendix I. Once the server is set up and the build passes, we can start developing the features.

Basically the workflow is the same as in section 4.2, but in addition to the tests passing on the local computer, or in *private build*, the tests must pass in continuous integration server. Also if there are acceptance tests in place, those must pass too.

If the project is much larger than the example below, the private build might contain only the unit tests, and the acceptance or regression tests are run as a part of the continuous integration build. To simulate this behavior, we'll create a few acceptance tests that are separated from the unit tests. These tests are not run as a part of the private build, but only in the CI server.

Once the feature has been developed, the acceptance tests have been enabled and the CI build passes, the CI server could automatically update the feature to the agile backlog management tool. Since there are no such existing backlog management tools, the behavior needs to be mimicked.

4.4.1 Creating the Acceptance Tests

First we will define the acceptance tests (for a full listing of the tests, see Appendix II):

- it "should return nil if no cans are loaded"
- it "should return nil if not enough money is inserted"
- it "should receive coins, drop a can and deposit coins"

These tests could have been agreed on and even written long before the feature is developed, but they will be enabled in the CI loop only when the feature is ready for acceptance testing. This will also ensure that the latest builds will always verify the correct functionality of that feature.

For the demonstration purposes, the acceptance tests have been written to "spec/acceptance/" folder and they are run only with the command "rake acceptance". The acceptance tests are presented completely in the Appendix II, below is presented only the essential parts of the tests:

```
describe "As a customer I want to buy a cold beverage using coins so
that I could quench my thirst" do

  before :each do
    @vending_machine = VendingMachine.new
  end

  describe "select and drop" do

    before :each do
      @coke_can = SodaCan.new(:coke)
      @coke_can.price = 1
    end

    it "should receive coins, drop a can and deposit coins" do
      @vending_machine.insert 1

      @coke_can = SodaCan.new(:coke)
      @coke_can.price = 1

      @vending_machine.load_cans @coke_can

      @vending_machine.select_and_drop(:coke).should == @coke_can
      @vending_machine.inserted_coins.should == 0
    end
  end
end
```

4.4.2 Continuous Integration Build

Once the features have been implemented (see Appendix II or the source code from github) and the new code + the acceptance tests have been committed to the version control, the continuous integration Build should start automatically:

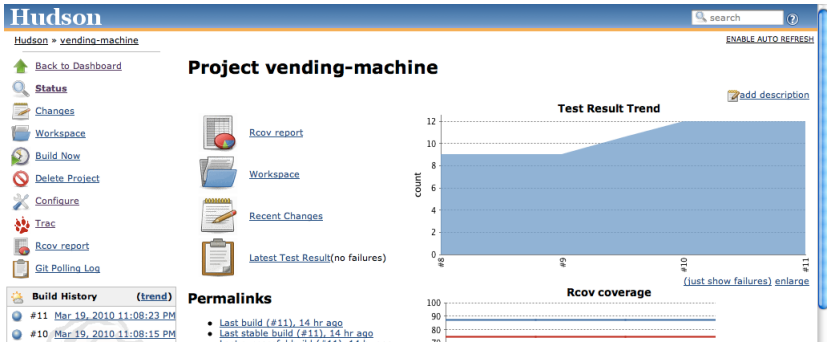


Figure 4-9: Continuous integration build containing the acceptance tests

4.4.3 Setting the Task Done Automatically

If the continuous integration Build passes, it could automatically update the task to the backlog management system, namely Agilefant. This feature has not been properly developed to any of the competitors, even though there are some existing plugins to e.g. Trac, Jira, Rally and Mantis. The existing plugins mainly create links between the *build* in the CI server and the *task* in the project management tool, although they should additionally update the status as *Done* in the project management application once the build passes successfully.

If the task completion were automatic, instead of logging back into the backlog management tool after finishing the task, it's enough that the developer commits her code with a certain *commit message format* into the version control system. The continuous integration Server notices the change and starts an automatic Build. If the build is successful, the task is automatically marked as *Done* in the tool.

For Mantis there exists an alpha version of a plugin, which updates the status automatically to the tracker after a successful build [23]. The developer uses a predefined commit message format, containing the id (000001):

```
$ git commit –am "fix issue 000001 Created acceptance tests \
                  for buying a can of coke"
$ git push origin master
```

And after the continuous integration server has noticed the change and successfully built the project, the build report contains a link to the tracker (Figure 4-9):

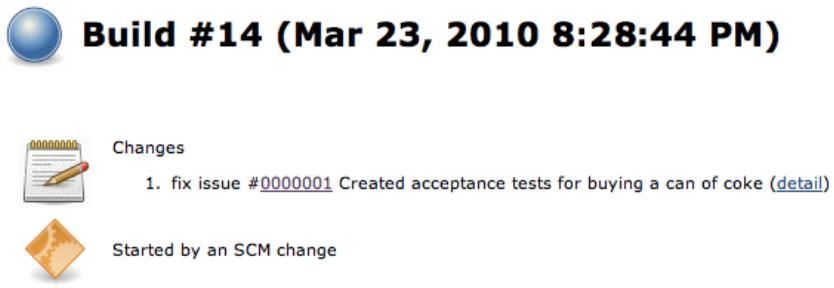
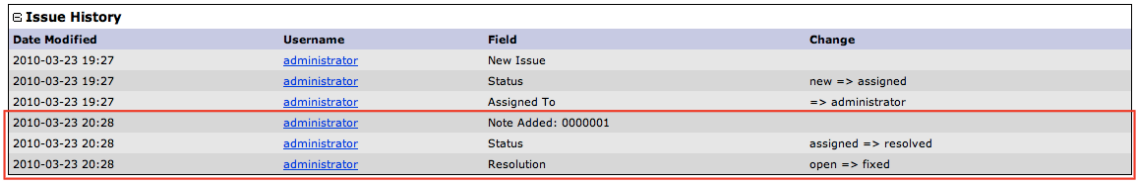


Figure 4-10: Link pointing to the Mantis bug tracker

And the tracker shows that the issue has been fixed (Figure 4-10):



Issue History

Date Modified	Username	Field	Change
2010-03-23 19:27	administrator	New Issue	
2010-03-23 19:27	administrator	Status	new => assigned
2010-03-23 19:27	administrator	Assigned To	=> administrator
2010-03-23 20:28	administrator	Note Added: 0000001	
2010-03-23 20:28	administrator	Status	assigned => resolved
2010-03-23 20:28	administrator	Resolution	open => fixed

Figure 4-11: Automatically updated status in Mantis tool from Hudson

This functionality is illustrated in Figure 4-11. The only drawback to this technique is that the developer must know the *identifier* of the task he's currently developing. If he had to log in to the backlog management tool to check the id, the whole process would be practically useless since the manual steps would not decrease.

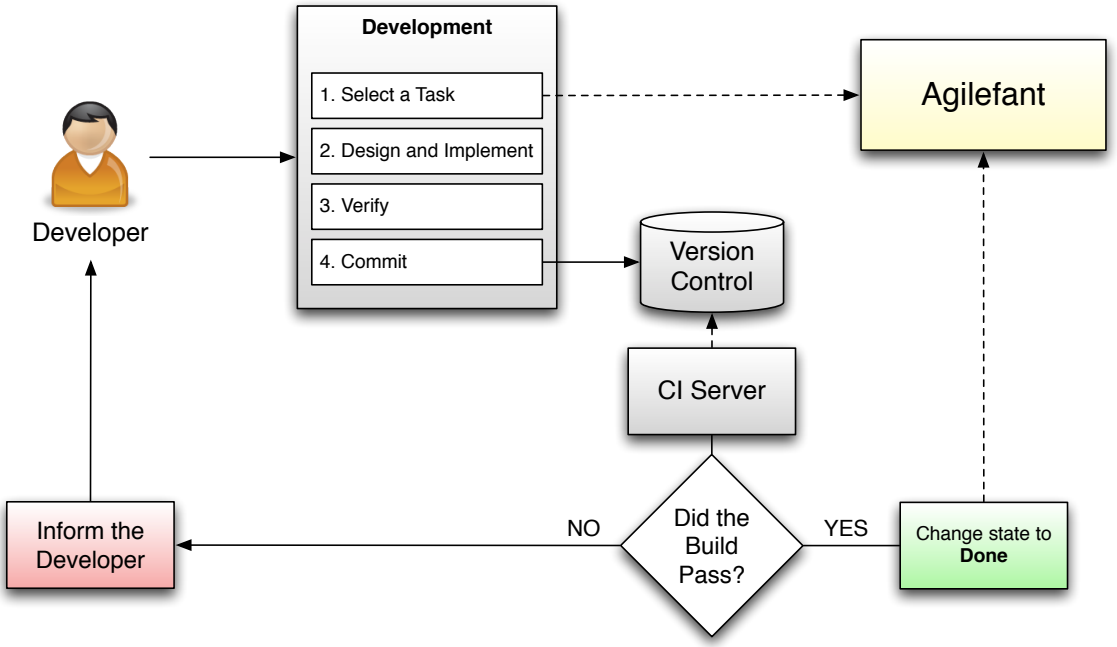


Figure 4-12: Manual Development with Automatic Task Acceptance

On the other hand the team could agree the acceptance tests together and if the User Stories behind the acceptance tests match with the ones in Agilefant, the CI Server plugin and Agilefant could automatically agree on the correct identifiers according to the user story. The developer could for example write a part of the User Story to the commit message, and the tools could find the correct story automatically.

The developers could also add a metadata identifier of the Agilefant story id to the acceptance test. If the tests are written before the implementation, the tests could already contain the story id in Agilefant. In Java, this could be added as a Javadoc metadata:

```
/**
 * @story 000001
 */
@Test
public void testShouldReceiveCoinsDropACanAndDepositCoins() {
    ...
}
```

It wouldn't matter what the developers wrote to the commit message, since the story id or ids would be interpreted from the metadata information. This behavior could be easily extended to any other test framework.

Developing a Task using Eclipse, Mylyn and continuous integration

If the developer used Eclipse and Mylyn for the development, the problem of knowing the correct task or story id disappears. The developer can update the status straight from Eclipse after verifying that *the unit tests pass*, *all of the functionalities have been properly implemented* and the *code has been committed* to the version control repository. The developers could this way benefit from the status division in the Agilfant tool: Developers set the task *Implemented*, and the CI Server changes it to *Done* after the build containing the Acceptance Tests passes.

Eclipse + Mylyn combination should even support automatic commit messages [24], [25] and [33], although that functionality is still very young and has not been implemented to any of the backlog management tools. When using the task-based development with Mylyn, the default format of an automatic commit message is:

```
${task.status} - ${connector.task.prefix} ${task.key}:
${task.description}

${task.url}
```

The suggested development workflow using this combination is illustrated in Figure 4-10.

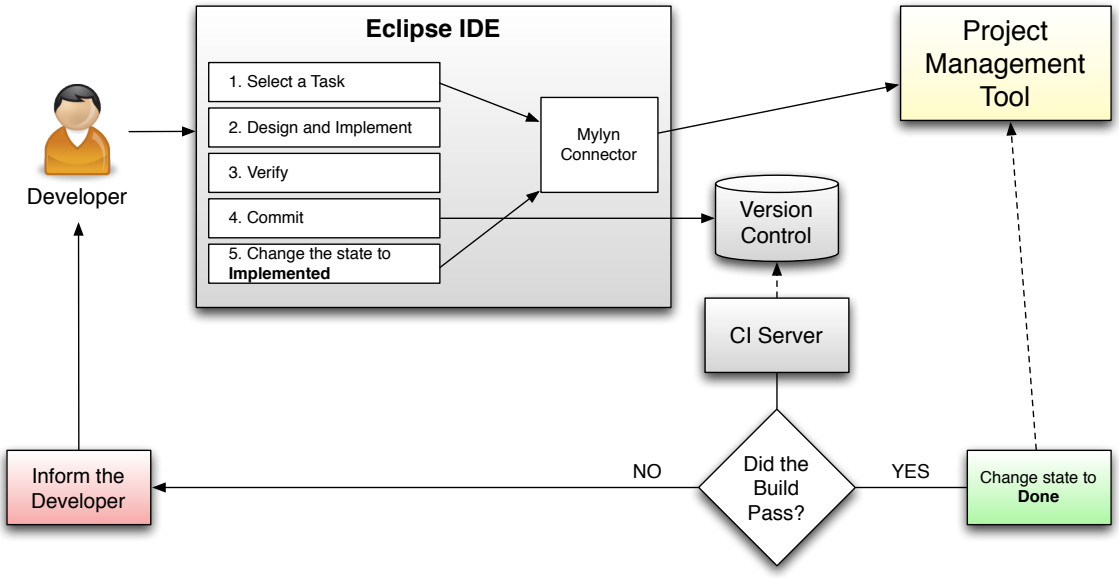


Figure 4-13: Development Workflow using Eclipse, Mylyn and continuous integration

5 Model Evaluation

The previous chapter presented four different ways of developing software using an online project management tool. Optimally the developers' results were verified automatically with a set of Acceptance Tests that have been agreed with the customer on a conceptual level. These acceptance tests are enabled separately for each of the developed feature or user story and are run either as a part of the developer's *private build* or an automatic continuous integration build. The user story statuses are updated to the project management tool automatically.

The major problem seems to be the connectivity between different tools. When developing the code completely manually (see section 4.2), it is completely the developer's responsibility to make sure that the feature actually has been developed, that it's working properly (i.e. the tests pass) and that the online backlog management tool has been updated accordingly. Once the automation level is increased (see sections 4.3 - 4.5), the developer runs into various connectivity problems. Either he has to know the issue number or identification from the project management tool (see section 4.4) to let it be marked *Done* automatically, or he has to have a tool connection like Eclipse and Mylyn already set up for the backlog management tool (see section 4.3).

When developing with high-level programming languages like Ruby or Python, it seems to be beneficial to just enable the acceptance tests for the continuous integration build. Once the developer sees it pass he can either set the task *Done* in the backlog management tool interface or e.g. ask another team member to do a short exploratory test session before closing the task (i.e. setting it as *Done*). For Java development this could be improved slightly with the Mylyn Connector, which enables the task view in the development tool. However the Mylyn is at its best when used only manually: The developer checks the continuous integration build result and sets the task in Mylyn to *Done* by hand after a passing build.

5.1 Small Scale Build Environments

The more the developer has tools, the more fragile the development environment becomes. The author of this work ran into various problems when setting up the environments. And once they were all properly set up, there was always something that prevented the full implementation of the toolset – further leading to doing exactly as much work as with the manual approach (section 4.2).

5.2 Large Scale Build Environments

If the build environment was large, i.e. the acceptance tests are not run as a part of the developer's *private build* and the generated product (from a continuous integration build) is further tested in some QA organization, there could be a need for backtracking the bugs to specific revisions of the code (see section 4.4). For this purpose a continuous integration plugin that connects the tasks or issues to the specific build version could be feasible. This would also be rather simple to implement, since it only displays the relations between the project management tool and the CI build, and such plugins already exist [26].

Nonetheless, this will not help with the tool selection. The developers would still have to update the task relationships between the version control and the project management tool. Eclipse + Mylyn (see section 4.3) gets closest to solving this problem; there is even a possibility for automatic version control messages [24]. However the author was not able to make that process even nearly as easy as just checking the task number from the Mylyn tool (or straight from Agilefant) and adding it to the version control commit message manually. This procedure is in any case too complex to be used - unless its absolutely necessary for the project organization and the build environment.

6 Tool Development

This chapter describes briefly the starting points for developing the automation interfaces for Agilefant. The major issue in the tool development is that the Agilefant WebService-interface is still in incubation, and the plugins cannot be tested, nor finished, at least with the current version of Agilefant.

6.1 Hudson CI Plugin for Agilefant

Developing the Hudson plugin itself is rather simple. The process has been defined [27] and there are many similar plugins, which all are open source products [26].

For example the Trac has an XML-RPC interface, which has an existing documentation [28]. Additionally the Trac community has created clients to that interface for different platforms, e.g. the ruby interface has a plugin called trac4r [29]. Also the Hudson plugin sources are accessible to anyone [30].

6.2 Mylyn Connector for Agilefant

The Mylyn connector is also pretty simple to implement if the project management tool's web service interface is well defined. Mylyn supports natively the same XML-RPC interface as for example Trac, and there are plugins for Jira and Trac, which are already implemented and are developed on open source basis [31].

7 Conclusions

This work has introduced the concept of automatic status update of a task using both IDE tools and continuous integration and thus answered the research questions *1. How the development workflow enables automatic task update* and *2. How the tasks could be automatically updated?* (See chapter 1.2 for research questions) These solutions were built upon existing tools and they were documented in chapter 4.

The objectives of this study also included the feasibility study of the automatic task update (research question *3. When is the automatic task update feasible?*) and based on this work the tools that are needed for doing automatic task updates are not mature enough for commercial software development. However the tools are constantly being developed towards easy and robust development environment, and probably one day the developers can combine a development IDE, online backlog management tool (Agilefant), and continuous integration together.

The last research question: *4. How to implement the automatic task update functionalities in Agilefant* was briefly discussed in chapter 6. Basically the tool implementations can be done easily by following the existing solutions, but the more arduous problem is that the Agilefant tool does not yet support any WebService interfaces, which in the end enables the interaction between tools.

8 Discussion and Future Work

8.1 Reliability and Validity of the Research

This work has been a quick glance at the task automation concept using a backlog management tool together with development IDE connector and the continuous integration practice. It should be viewed as a preliminary vision of the topic for further research and concept development. The conclusions have been made by the author and therefore there is much room for future discussions especially in the open source community developing these tools further.

8.2 Applicability of the Results

The concepts presented in chapter 4 have all been implemented using the authors existing experience of software development in commercial companies. Therefore the development methods can be used as such, at least to the extent where the existing tools make it possible. Even if the IDE tools or programming languages are different, the continuous integration practice and using the Agilefant backlog management tool still can be implemented using the methods presented in this work.

8.3 Suggestions for Further Research

Since most of the tools presented in this work are being developed with accelerating pace by the open source community, it is important that the results from this work are constantly reviewed and re-evaluated. It is almost sure that the concepts that were too complex for implementation according to this study will become feasible one day.

According to this study the Agilefant tool desperately needs an open two-way interface for connecting other applications. The suggested method according to this work is using the XML-RPC library from Apache [32].

Appendix I: Setting it All Up

This chapter contains the installation instructions for various tools used for the actual development presented in this work.

Installing Ruby

If Ruby programming language is not yet supported by your Operating System, go to http://www.ruby-lang.org/en/downloads/ to install the latest version on any platform.

Installing Eclipse and Tools

Installing Eclipse is best described here: http://www.eclipse.org/downloads/

Installing the Eclipse Mylyn Connector can be found from here: http://www.eclipse.org/mylyn/downloads/

The Subversive SVN -plugin is distributed as a supplemental part of the Eclipse distribution. The clearest installation instructions at the time of writing have been written by Ben Christensen in his blog: http://benjchristensen.com/2009/06/24/eclipse-galileo-3-5-and-subversion/

Installing continuous integration Server and Projects

The continuous integration concept is presented in Chapter 2.5, and the information provided by a continuous integration server is shown in Figure 4-8.

Downloading and starting the Server

- Download Hudson: https://hudson.dev.java.net/
- Start it up by running the following command

```
java –jar hudson.war
```

- Open your browser to http://localhost:8080/

Adding the Necessary Plugins

- Add the Git version control support by installing the plugin at **Manage Hudson** -> **Manage Plugins** -> Choose tab **Available ->** Select:
    - Select **Git Plugin**
    - Select **Rake plugin**
    - Select **Ruby Metrics Plugin**
    - Click **Install**
- Restart the Hudson server from the Shell

Creating the Ruby Project

For the **Ruby**-based Build reporting you will additionally need a plugin called **ci_reporter**. This is installed in the shell by typing:

```
sudo gem install ci_reporter
```

Add the Ruby Example Project to the Hudson by following these steps:

- In the front page of the server click **New Job**
- Type "*vending-machine*" as the **Job Name**
- Select **Build a free-style software project**
- Click **OK**

- Go to section **Source Code Management**
    - Select **Git** and add the repository location: git@github.com:mrako/vending-machine.git
- Go to section **Build Triggers** and select **Poll SCM**
    - Type "* * * * *" to **Schedule** to check for changes in the repository every minute.
- Go to tab **Build**
    - Add build step **Invoke Rake** and type "*ci*" to the **Tasks** –field
- Go to **Post-build Actions**
    - Select Publish **JUnit test result report** and type "*spec/reports/*.xml*"
    - Select Publish **RCov report** and type "*coverage*" as the **Rcov report directory**
    - Select Publish **Rails stats report**
- Click **Save**

- Now the build can be started by clicking **Build Now**

Creating the Java Project

Add the Java Example Project to the Hudson by following these steps:

- In the front page of the server click **New Job**
- Type "*vending-machine-java*" as the **Job Name**
- Select **Build a free-style software project**
- Click **OK**

- Go to section **Source Code Management**
  - Select **Subversion** and add the repository location: https://Agilefant-connector-example.googlecode.com/svn/trunk/vending-machine/java
- Go to section **Build Triggers** and select **Poll SCM**
  - Type "* * * * *" to **Schedule** to check for changes in the repository every minute.
- Go to tab **Build**
  - Write **test** to *Goals and Options* -field
- Click **Save**

- Now the build can be started by clicking **Build Now**

Appendix II: The Acceptance Tests in Ruby

```
require File.dirname(__FILE__) + '/../spec_helper'

require 'vending_machine'
require 'soda_can'

describe "As a customer I want to buy a cold beverage using coins so
that I could quench my thirst" do
  before :each do
    @vending_machine = VendingMachine.new
  end

  describe "select and drop" do

    before :each do
      @coke_can = SodaCan.new(:coke)
      @coke_can.price = 1
    end

    it "should return nil if no cans are loaded" do
      @vending_machine.select_and_drop(:coke).should be_nil
    end

    it "should return nil if not enough money is inserted" do
      @coke_can = SodaCan.new(:coke)
      @coke_can.price = 1
      @vending_machine.load_cans @coke_can

      @vending_machine.insert 0.5

      @vending_machine.select_and_drop(:coke).should be_nil
    end

    it "should receive coins, drop a can and deposit coins" do
      @vending_machine.insert 1

      @coke_can = SodaCan.new(:coke)
      @coke_can.price = 1

      @vending_machine.load_cans @coke_can

      @vending_machine.select_and_drop(:coke).should == @coke_can
      @vending_machine.inserted_coins.should == 0
    end
  end
end
```

References

[1] Poppendieck M. and T. (2003). *Lean Software Development: An agile Toolkit*. Addison-Wesley Professional.

[2] Beck K. (1999). *Extreme Programming Explained*. Addison-Wesley Professional.

[3] Schwaber, K., Beedle M. (2001). *agile Software Development with Scrum*. Prentice Hall.

[4] Cohn M. (2005). *agile Estimating and Planning*. Prentice Hall.

[5] Cohn M. (2009). *Succeeding with agile: Software Development Using Scrum*. Addison-Wesley Professional.

[6] Cohn, M. (2004). *User Stories Applied: For agile Software Development*. Addison-Wesley Professional

[7] Davies, R. (2001) *The Power of Stories*. Practitioners report/poster presentation

[8] Cohn, M. (2008). http://blog.mountaingoatsoftware.com/advantages-of-the-as-a-user-i-want-user-story-template

[9] Pichler, R. (2010). *agile Product Management with Scrum: Creating Products that Customers Love*. Addison-Wesley Professional.

[10] http://agilefaq.net/2007/10/24/what-is-definition-of-done/

[11] http://www.martinfowler.com/articles/continuousIntegration.html

[12] Duvall P. (2007). *continuous integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional

[13] Patton R. (2005). *Software Testing, Second Edition*. Sams.

[14] Koskela, L. (2007). *Test Driven: TDD and Acceptance TDD for Java Developers*. Manning Publications

[15] http://blog.xebia.com/wp-content/uploads/2008/09/task-board-no-toploading.png

[16] http://www.Agilefant.org/

[17] http://www.eclipse.org/

[18] http://www.eclipse.org/mylyn/

[19] http://www.rallydev.com/

[20] http://git-scm.com/

[21] https://github.com/

[22] http://code.google.com/

[23] http://wiki.hudson-ci.org/display/HUDSON/Mantis+Plugin

[24] http://www.easyeclipse.org/site/plugins/subclipse-mylyn.html

[25] http://www.ibm.com/developerworks/java/library/j-mylyn2/

[26] http://wiki.hudson-ci.org/display/HUDSON/Plugins

[27] http://wiki.hudson-ci.org/display/HUDSON/Plugin+tutorial

[28] http://trac-hacks.org/wiki/XmlRpcPlugin

[29] http://github.com/csexton/trac4r

[30] https://hudson.dev.java.net/svn/hudson/trunk/hudson/plugins/trac/ (username: *guest*, password: <blank>)

[31] http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.mylyn/?root=Tools_Project

[32] http://ws.apache.org/xmlrpc/

[33] http://wiki.eclipse.org/Mylyn_User_Guide#Automatic_Commit_Messages

[34] http://code.google.com/p/robotframework/

[35] http://fit.c2.com/

[36] http://cukes.info/

[37] http://wiki.hudson-ci.org/display/HUDSON/Hudson+Build+Status+Lava+Lamps

[38] http://code.google.com/p/hudson-eclipse/