Coursework

Team Coursework Exercise 2
Working with Features

COMP23311- Software Engineering I

University of Manchester

School of Computer Science

2018/2019

written by

Suzanne M. Embury

1 Overview

In this second exercise using the Stendhal code base, you and your team are again presented with a number of issues recorded in your GitLab issue tracker. This time, however, the issues describe new features rather than bugs. That is, they describe extensions to the Stendhal game, rather than corrections to the implementation that bring it back into line with the way the game should behave.

This brings into play a number of new software engineering skills, in addition to the skills for small-scale change we looked at in the first few weeks of the course unit. The first difference is the amount of work to be done. Unlike the simple bugs we looked at in the last coursework exercise, the features your team has been presented with vary in size, with some simple and straightforward and some more complex, involving the modification of existing classes and the design of new classes. Taken as a whole, they represent more work than you can sensibly achieve in the time given for the coursework. **You must therefore choose a subset of the issues that your team will commit to deliver by the deadline.**

Another difference between the implementation of new features and the fixing of bugs is in the scope of the work. Most of the bug fixes we looked at were so simple, they were implemented in full, with few decisions to be made about how much of the work described by the issue it was worthwhile to complete. When adding new features, the situation is more complicated: we can choose to add a simpler or more complex and involved version of the feature. *For this coursework, you are asked to implement the simplest playable version of any selected features for inclusion in your release.* The version you choose should minimise the risks of regression, but should also be extensible to a fuller version of the feature in future, without introducing unpleasant surprises for players.

This exercise will follow the same broad phases as the first one, with one additional step:

- Step 1: a planning phase.
- Step 2: a testing and implementation phase.
- Step 3: a code review phase.
- Step 4: an integration and system testing phase.
- Step 5: a release phase.
- Step 6: a preparation phase for the face-to-face interviews.

You can choose how you carry out these steps in your team, perhaps overlapping some of the steps, or carrying out the whole process for individual issues separately.

The exercise will test your ability to:

- use basic software estimation techniques to commit to a workload that your team can deliver, and to divide work fairly across the team,
- scope features to control the risk of adding new functionality to an existing development,
- make use of a simple Git workflow for coordinating team development,
- write test suites that maintain test coverage across new and existing code,
- use code reading techniques to locate the parts of a large software system where new functionality can be safely added,
- use basic code review techniques to help ensure the quality of your team's code, and
- use test-first development and good OO design principles to guard against regression in this and future releases.

2 What You Have to Do

Step 1: High Level Planning

Your first task is to decide which of the issues you will commit to including in the next release, how you will scope them down into minimum marketable features and who will be responsible for which parts of the work.

You should create basic work-breakdown structures (WBS) to come up with defensible estimates for the effort needed to implement each issue.

Bearing in mind the time you have available in which to complete the coursework (the team study sessions and your personal study time for 2 and a half weeks) and the skills you have in your team, you should use these estimates to decide:

- which of the issues you are going to implement,
- which team member will be the lead for each issue,
- which team members will work on the implementation of each issue, and
- which team members will act as code reviewers for each issue.

All these decisions should be documented in your issue tracker and GitLab project wiki.

Lead team members should be assigned as the responsible person for the issue in the issue tracker. This person is responsible for the detailed planning for their issue, for scoping it down, for coordinating the implementation work amongst the sub-group assigned to it, and for documenting the work as needed for marking.

Sub-team members for each issue must be recorded on the issue tracker, using a sentence of the form:

> Sub-team member: <link to GitLab profile of team member>

GitLab profile links have the form: `https://gitlab.cs.man.ac.uk/name`, where `name` is a GitLab username. Multiple sub-team members may be recorded in separate comments, or in a single comment with newlines between the sentences. You do not need to record the team lead as a sub-team member.

Code reviewers for issues must also be recorded on the issue tracker, using a sentence of the form:

> Code review by: <link to GitLab profile of team member>

Multiple code reviewers may be recorded for a given issue, if wished.

You should document your estimate for the **effort** required to implement each feature in the issue tracker. For this exercise, we are going to use GitLab's Time Tracking facility. Information about how to do this can be found at:

`https://gitlab.cs.man.ac.uk/help/workflow/time_tracking.md`

You should document the WBSs you create for each issue at this stage on your team's GitLab wiki. Create a page with the following page slugs for each issue.

- Add Rentable Allotments to Semos Fields: `rentable-allotments`
- Wheelbarrows: `wheelbarrows`
- Lon Jatham at the University of Manchester Ados Campus: `lon-jatham`
- A Teleportation Scroll for Wofol City: `wofol-scroll`
- Adventurers' Guild: `adventurers-guild`
- An Invisibility Ring: `invisibility-ring`

Use the text after the colon as the page slug when creating the page for each issue. If done correctly, this would mean that the wiki page for the Lon Jatham issue would be reached at the following URL, for team 47:

`https://gitlab.cs.man.ac.uk/comp23311_2018/stendhal_S1Team47/wikis/lon-jatham`

The wiki page for each issue should include the basic WBS you create in order to estimate the difficulty of the issue.

It should also contain a brief justification for your decision to implement this feature or not. (Two or three sentences of justification is sufficient.)

Finally, once you have decided which features you will not be delivering in this release, you should remove them from the Coursework 2 Milestone. Only the issues you are committing to deliver should be associated with this milestone by the deadline for the exercise.

Step 2: Testing and Implementation

Once features have been allocated a team lead, feature teams can start work on the implementation of their features. The team lead should begin by creating a detailed WBS for the work on the issue's wiki page, and by allocating parts of the work to the sub-team members. Care should be taken to find a breakdown of the work that allows sub-team members to work in parallel on different parts of the feature.

The team lead should set a due date for the delivery of the feature to the team, using the issue tracker's Due Date facility. Obviously, this due date should be chosen to allow time for code review, corrections and merging of the feature.

The team lead should update the estimate for the issue in the issue tracker if, based on the more detailed planning carried out, the sub-team feel the original estimate was inaccurate.

We ask you to use the same Git workflow as you used in the first exercise, to allow your sub-team to work collaboratively on each issue without interfering with the work of the rest of your team.

However, there are some new skills we ask you to demonstrate in this second course unit. We are asking you to control the size and risk of the work, using feature scoping, and to code in a test-first manner. We are also asking you to manage the quality of the commit messages your team makes, and to use a test coverage tool to assess whether you have written sufficient tests or not. Details of all these are given below.

Choosing a Sensible Starting Commit

It is important that all the team start their feature branches from a clean commit. Otherwise, problems in earlier commits (such as failing tests) will be carried forward into the new feature branches, and may result in lost marks for exercise 2.

Teams with a clean build for their development branch in Jenkins should be okay to start the work for the next coursework exercise from this point. Teams with unstable or failed builds on their development branch will need to start by getting the `master` branch to a clean state. If the problems were caused by merging feature branches with unstable or failed merges into `master`, then you will need to revert those commits[1]. Please get help in an early team study session if you are unsure about this

Once a suitable starting commit has been identified or created, one member of your team should create a tag pointing to that commit. The tag should be called:

```
COMP23311/EX2/starting_point
```

We have created a Jenkins build for this tag, to help you choose a good starting point for your work for this exercise. Push the tag to your candidate starting commit and then manually request that the job be built (You'll need to build this job manually if you push this tag without code changes.)

If you place the tag at the wrong commit, and need to change it, you will need to delete it and recreate it. Instructions for deleting tags can be found at the end of this document.

It is expected that your development branch will start at this commit for exercise 2, so make sure your master branch is located at this commit before you start work on the features. All feature branches for exercise 2 should begin at this tagged commit or a later commit.

Git Workflow

As before, you should use feature branches to keep the work on each feature separate from the main development branch, until you are satisfied that the code is ready to be integrated into the main game code. Please continue to follow the Stendhal team's practice of using the `master` branch as the development branch.

You must use the following feature branch names:

- Add Rentable Allotments to Semos Fields: `COMP23311/EX2/rentable-allotments`

[1]The use of Git revert as a quick fix for Git errors is never pretty and should be avoided where possible. However, in this case, it may be the quickest way to get your codebase to a clean starting state for the exercise.

- Wheelbarrows: `COMP23311/EX2/wheelbarrows`
- Lon Jatham at the University of Manchester Ados Campus: `COMP23311/EX2/lon-jatham`
- A Teleportation Scroll for Wofol City: `COMP23311/EX2/wofol-scroll`
- Adventurers' Guild: `COMP23311/EX2/adventurers-guild`
- An Invisibility Ring: `COMP23311/EX2/invisibility-ring`

These branch names are expected by the automated marking system. Failure to use these exact names will mean that the marking system is unable to find and award marks for your work.

Best Practice Commit Messages

For this exercise, we ask you to gradually increase your Git skills by following best practice in terms of the commit messages you use. We ask you to adopt the style of commit message recommended by GitHub. Details of this format can be found on Blackboard[2].

An example of a commit message that follows the format we want you to use is:

```
Allow Joshua to list ingredients for spy glass

When asked, Joshua will now list the ingredients he needs for making
the new spy glass item.  This was trickier than it looked, because
Joshua was already using all the available conversation states.  I
added a new state (QUESTION_4) to make it work.
```

Please note that all team members must use their own GitLab account to commit and push their work, and that commits must have your University e-mail address as the e-mail of the commit author. This is set using `git config`, as described on the School wiki:

`https://wiki.cs.manchester.ac.uk/index.php/Gitlab/Git_Setup`

Please make sure you set the `user.name` and `user.email` variables on all machines you intend to commit from.

[2]Under `Course Content`→`Week 7` and `Assessment`→`Team Coursework`→`Exercise 2`→`Related Documents`.

Test-First Coding

As in exercise 1, you are asked to write tests to describe the behaviour of the code change you are making. For this exercise, we ask you to use the test-first development strategy introduced in the workshops.

Each sub-team should begin by writing at least two failing acceptance tests for the feature you plan to implement, following the test-first approach demonstrated in the workshops. These tests should describe core elements of the features, not trivial side cases. For example, a test where a quest is undertaken and completed in the quickest way would be considered core, while a test where a quest is refused at first offer is not.

To help us assess and give feedback on your use of test-first development, we ask you to commit these two failing acceptance tests **before** you commit any of the production code changes that will make them pass, in a single commit. You may make changes to these tests in later commits, as well (of course) as changes to production code. But the first commit should contain two tests that are sufficiently complete in their implementation to allow the code to compile and run to an unstable build. (This is not normal practice, but is something we are asking you to do in this exercise, to show that you understand the principle of test-first coding.)

Managing Test Coverage

When writing tests for more extensive functionality changes, you may wonder when you have written enough tests and when you can stop. This is a critical question for software developers, since writing tests is not free (neither in terms of developer time, nor the time required to run the tests). For this coursework, we ask you to use test coverage to help you manage your tests, and determine whether you need to write more.

Since this is the first time that most of you have used a test coverage tool, we are setting a fairly low coverage goal for this exercise. We ask you to ensure that you maintain the overall instruction coverage level for the coursework set by the Stendhal team in the starting commit for the coursework. So, your coverage goal for this exercise is:

51% instruction coverage for all classes

We will mark your team's coverage using the instruction coverage for all classes, given in the "Overall Coverage Summary" values on your team's development branch build in Jenkins.

If your team's coverage is lower than this value, you will need to use the coverage reports to find areas of the new code you have written that is not well covered by tests. Looking at the instructions that were not executed during the test suite execution, you'll need to think of tests to add that will cause those instructions to be executed.

Documenting Your Work

You should use the issue tracker and/or the team wiki to document any problems you encounter while implementing that require you to make changes to your plan or your team structure. Discussion while implementing using other tools (such as Facebook or Slack) is obviously fine but remember that we can only mark what we can see. Key decisions regarding who does what work, and changes to the planned scope for features that are running late, must be documented on your GitLab repository if we are to be able to take them into account when marking.

An important piece of documentation to keep is the record of how long you have spent on the feature so far. You should record this in your team's issue tracker on GitLab, using the "time tracking" feature, as you did in exercise 1. Information on how to use this is available from the side bar of each issue. Note that you will need to record all the time spent by all sub-team members. You will probably find it easier to update the time spent as you go along, rather than trying to remember and adding the total in at the end.

Step 3: Code Review

For this exercise, we are asking you to start to make use of an additional code quality management technique: code reviews. Your goal is to ensure that no code is merged into the development branch without having an independent team member look over it and check for errors or code quality problems. You will lose marks if unreviewed code is merged into the development branch.

Both test code and production code changes must be reviewed. You can choose whether to review individual commits as you proceed, or to review the whole code for a branch before merging.

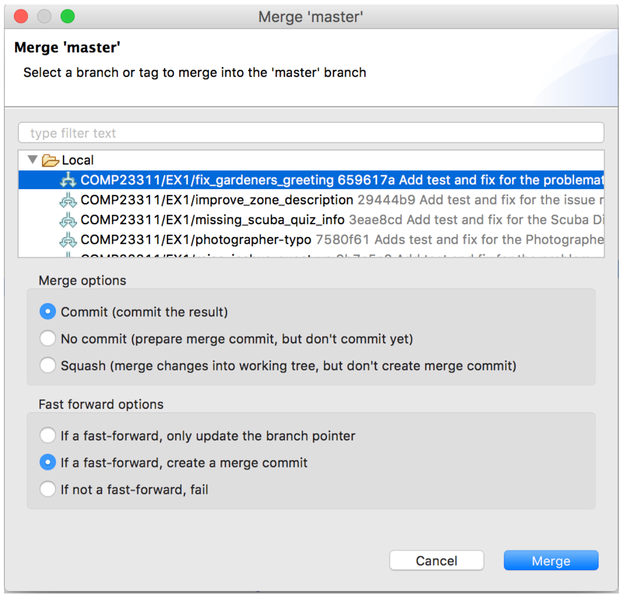
Reviews must be given using the GitLab commit comment feature, or through merge requests. Reviews given verbally, or recorded in some tool other than GitLab, will receive no marks (for obvious reasons).

Every contributing team member must perform at least one review in GitLab across the exercise.

A guide to performing code review can be found under `Course Content`→`Week 7` on Blackboard.

Step 4: Integrate Completed Features into the Development Branch

When you are satisfied that a feature branch contains code that is fit to be integrated into the game, you should merge the feature branch into the development branch. For this exercise, **we ask that you use non-fast-forward merges for all merges of your feature branch into the development branch**[3]. To do this in Eclipse, use the `Team→Merge` command from the project context menu, and select the option to create a merge commit in the event of a fast-forward merge:



Merge 'master'

Merge 'master'

Select a branch or tag to merge into the 'master' branch

type filter text

Local
- COMP23311/EX1/fix_gardeners_greeting 659617a Add test and fix for the problemat
- COMP23311/EX1/improve_zone_description 29444b9 Add test and fix for the issue r
- COMP23311/EX1/missing_scuba_quiz_info 3eae8cd Add test and fix for the Scuba Di
- COMP23311/EX1/photographer-typo 7580f61 Adds test and fix for the Photographe

Merge options
- [✓ Commit (commit the result)]
- No commit (prepare merge commit, but don't commit yet)
- Squash (merge changes into working tree, but don't create merge commit)

Fast forward options
- If a fast-forward, only update the branch pointer
- [✓ If a fast-forward, create a merge commit]
- If not a fast-forward, fail

Unfortunately, the handy merge option accessible through the History View doesn't give the chance to request this option, so should not be used for merges of your feature branch into the development branch. If you use this version of merge by mistake, don't panic. Fast forward merges are easily undone just after you make them by resetting master to the commit it was at before the merge was made *provided you have not pushed your merge to your team's remote*. Please continue to make merges locally, checking them in the History View before carrying out the merge in your team's remote.

You may use merge requests for this coursework exercise, should you wish, to provide a "quality gate" for code, to prevent it from being integrated into the development branch before it has undergone code review. You may also perform the merge itself through merge requests, but you are advised to use caution if you do. It is very easy to create an incorrect merge using merge requests unless you really understand how they work. Worse still is

[3]For this coursework, you are asked not to use rebase when integrating feature branch commits into the development branch. There are lots of good reasons to use rebase, but it really only makes sense when used in conjunction with Git features for creating a tidy commit history. Our goal at this stage in your degree is for you to be comfortable with the simpler merge approach. If you want to get some experience with rebase (a good idea), you can create a separate clone of your team's repository and hack it about to your heart's content, without putting your team's coursework mark at risk by trying to push any of the results.

that they create a merge directly into your team's remote repository, making it difficult to fix any merge errors that have been made. Even if you plan to create the merge through a merge request, we still recommend checking the merge out on a temporary local branch before pressing any GitLab merge buttons. (We've also set up Jenkins jobs to merge your feature branches into the development branch on each commit, to give you early warning of any problems that might arise—these builds are true "continuous integration" builds. The merges are carried out on Jenkins and are thrown away once the next build takes place. The merge will not be pushed to your team's repository.)

Once your feature branch has been merged into the development branch, and the resulting commit produces a clean stable build, you can consider the issue to be closed and should update its status in the issue tracker. Don't forget to add the time required to carry out the merge to the issue's time tracker.

Issues for un-merged features should be left open, even if some commits have been made for them. The team needs to know that this feature was not completed, so they can return to it in a future release should customer interest in the functionality proposed make the effort worthwhile.

Step 5: Prepare the Release

The final technical step is to prepare the release. Although several team members may contribute commits for this process, a single team member should take responsibility for making sure the release is created correctly. This team member should create an issue for this task called:

```
Prepare release 1.29uom
```

This issue should be associated with the coursework 2 milestone, and assigned to whichever team member is taking responsibility for carrying out the release task.

Choose the commit on the development branch which will form the basis for the release. This commit should include all the changes for the features that have been completed by the team during the coursework and have been merged successfully with the development branch. These are the issues that will be included in the release.

Once again, you will need to update:

- The version number of the software is updated (to 1.29uom).
- The `doc/AUTHORS.txt` file

- The description of the changes included in the release in the `doc/CHANGES.txt` file.

You can make these changes directly on the development branch (or you can use a feature branch and merge with the development branch when complete, if you wish).

When you have created a commit that contains all the code and documentation you want to release, you should mark this by adding a *tag* at that commit. The tag **must** be called:

```
VERSION_01_RELEASE_29_UOM
```

This is the version of the code that we will consider to be your released code, when we are marking. So, it is important that you place it at the right place. You can create the tag locally and push it, or you can use the GitLab web interface to create the tag once the release commits have been pushed.

Step 6: Prepare for the Marking Interview

You are done with the technical work at this stage, but there is one more task to do: prepare your team for the marking interview in week 9. In this interview, your TA will ask you to demonstrate some of your features in the released code, and will discuss with you how you have organised your work to balance load across the team and what monitoring steps you've taken to keep the work of the team on track for the deadline.

More information about the timing, location and format of this interview can be found on Blackboard, under `Assessment→Team Coursework→Exercise 2`.

Note that all team members must attend the marking interview, and any team member may be asked to demonstrate and talk about the issue they were responsible for. **Any team member who fails to attend and who has not registered a legitimate excuse with SSO or a member of the course team will receive an automatic 50% penalty on their mark for the exercise.** Team members who attend the interview will be unaffected by this penalty.

3 Submission of Your Team' s Work

All submission of work for this coursework exercise is through your team's GitLab repository. All you have to do is make sure the contents of your issue tracker, wiki, commit comments and Git repository are ready to be marked by the deadline. There is nothing else you have to submit.

Once the deadline is passed, an automated process will clone your repository and the state of your issue tracker. The automated process and the TAs will mark based on this clone, so no changes you make to the repository after the deadline will have any affect on your team mark.

4 Coursework Extensions

Since this coursework is a team exercise, no extensions will be given. Team members who experience substantial difficulties in completing their work due to illness or other legitimate reasons will need to complete a Mitigating Circumstances form so that this can be taken into account later. The marking process is sufficiently flexible to take into account non-contributing team members without significantly affecting the team mark.

If you are not going to be able to carry out the work for your issue by the deadline set for your team, you *must* inform the other team members in plenty of time. This will allow them to make decisions about what to include in the release, so they don't lose time dealing with the fact that your work has not been done.

5 Non-Contributing Team Members

Every team member is expected to contribute some meaningful code to the team's repository. Commits on feature branches should be made by the members of the sub-team recorded as responsible for the feature in the issue tracker.

Any student who has not made at least one meaningful commit to their team repository, from their School GitLab account, during the period covered by the exercise, will automatically receive a mark of 0 for the whole exercise.

This applies even if you decide to work in pairs on the issues. Sitting and watching someone else make a commit, even if you are telling them what to type, does not count as a commit from you. The commit must be made from your own GitLab account.

A meaningful commit is one that contributes code changes to either test or production code that moves the team's repository closer to the fix for an issue in some way. Adding white space, rewording comments or moving lines about are all examples of code changes that will not be considered to represent a meaningful contribution to the exercise.

6 Partially Contributing Team Members

If a team member contributes something, but does much less than others or contributes their work in a way that causes problems for the rest of the team, the team as a whole can choose to reduce the mark of that student. For this to happen, you must:

- Send an e-mails to the student as soon as the problem is noticed, pointing out the difficulties they are causing for the team, and asking them to say what they can do to resolve matters. CC this e-mail to Suzanne, so we have a formal record of the communication.
- Set a deadline for the team's work that is sufficiently far ahead of the actual deadline, so you have time to chase people who don't deliver.
- Before the team interview, send an e-mail to Suzanne *and* the offending team member letting them know that the team will propose a reduced mark for them at the interview.
- At the interview, raise the issue with the TA, who will document the circumstances on your marking form, along with details of the proposed mark reduction. If the affected team member agrees, the proposed reduction will be applied at that point.
- If team agreement on the mark reduction cannot be reached, the whole team will need to meet with Suzanne to agree a way forward.

Note that this process is not necessary for team members who have not made any commits in your team repository, as they will automatically receive a mark of 0 in this case.

Mark reductions apply to individual team members only. There is no effect on the mark of the rest of the team. Teams are asked to try to resolve problems within the team if possible, before making mark reductions, but this option is there as a measure of last resort for those teams who need it.

7 Plagiarism

This coursework is subject to the University's policies on plagiarism. Details can be found on the School web pages at:

`http://studentnet.cs.manchester.ac.uk/assessment/plagiarism.php?view=ug`

Note that committing the work of other people from your GitLab account counts as plagiarism, and action will be taken when it is detected. Rebasing commits authored by others does not count as plagiarism, providing the original authorship information is retained in the commit.

8 Technical Appendix

Deleting a Tag on GitLab

If you need to move the position of a tag you have created in GitLab, you will need to first delete the tag (using the red dustbin icon on the Tags page) and to recreate it in the right place.

This is easy enough. However, as always when making changes to a team's remote repository, there is a complication to be aware of. If other team members have the old tag in their local repository, you will need to make sure that the tag is updated to point at the right commit, before someone pushes the old location back to the repository again.

Therefore, as soon as the tag has been recreated in GitLab, **all** team members should run the following command, at the command line, from the folder where their local Stendhal Git repository is located:

```
% git fetch --tags
```

You can check that the tag now refers to the correct commit using the command:

```
% git rev-list -1 <tag name>
```

It is recommended that operations on your team repository, such as correcting the location of a tag, are done in a team study session, when all the team is present and able to carry out the necessary commands on their local repositories at the same time.