Coursework

Team Coursework Exercise 1
Dealing with Small Scale Code Changes

COMP23311- Software Engineering I

University of Manchester

School of Computer Science

2018/2019

written by

Suzanne M. Embury

1 Overview

For this first team coursework exercise, your team has been provided with a Git repository hosted on the School's GitLab server. This repository contains a slightly modified version version of the Stendhal code base. We've populated the issue tracker of the repository with a description of several "bugs" in the code that you must fix by the coursework deadline.

But we're not just asking you to fix the bugs. We are asking you to work like a professional software development team. You must ensure that all the issues you are fixing are covered by unit tests before you fix them, and you must use feature branches to protect the work of other team members from any mistakes you might introduce with your own team. We're also asking you to keep a close eye on the quality of your development and release branches (with the help of a continuous integration server), and to use basic Git techniques to ensure that, whatever happens to the code on your feature branches, only code changes that compile and pass the unit tests reach the release branch.

The exercise will test your ability to:

- make use of a simple Git workflow, based on feature branches, to allow multiple coders to work safely on a code base at the same time,
- use an automated test suite to make code changes to a large body of code without causing regression,
- write new tests to make bugs visible before you fix them,
- use code reading techniques to locate the parts of a large software system that are relevant to a particular bug, and
- prepare a good quality release incorporating the work of multiple developers.

2 What You Have to Do

Step 1

Following the same process covered in the workshops, each team member should acquire their own local copy of the code, learn how to build and test it, and run the code in their repository as a locally hosted version of the game. Instructions for doing this are provided on Blackboard, under `Course Content→Week 2→Tuesday Team Study`.

Step 2

Examine the issues in the issue tracker for your new repository. For each issue in this list, perform a minimal examination of the code base and decide as a team who will be responsible for fixing each bug. The issues should be assigned to the responsible person in the issue tracker.

Important: every member of the team must be assigned as the responsible person for one issue.

If your team has 6 members, then you should plan to fix all the issues. If your team has fewer members than that, you can choose which issues you will address. For example, a team of 5 students will choose 5 of the issues to address in this coursework, and may ignore the 6th. In this case, leave any unselected issues unassigned, and remove them from the Coursework 1 Release Deadline milestone in your issue tracker. The automated marking system will mark only those issues that are associated with the milestone at the deadline.

Step 3

Each team member should make an estimate of how long it will take to fix the bug they are responsible for. This estimate should be recorded using 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`

At this stage, it is not important that your estimate is correct. What is most important is that you make an attempt at estimating before starting to write code for the fix.

You should also set a Due Date for each issue in the issue tracker. This should be the date when the work on this individual issue should be completed by. Note that this *should not* be the deadline for the coursework. You need to complete your individual work on the issue in time to merge the work into the development branch and for the new release to be created.

Step 4

You must use separate feature branches for each issue, to protect your team mates from being affected by your changes until they are complete and ready for use. The next step, therefore, is to set up the branch for the issue you are responsible for fixing.

You must use the following branch names:

- For issue 1, use the branch name `COMP23311/EX1/nalwor-entrance-teleport`
- For issue 2, use the branch name `COMP23311/EX1/soup-quest-karma`
- For issue 3, use the branch name `COMP23311/EX1/ados-librarian`
- For issue 4, use the branch name `COMP23311/EX1/straw-carts-vanish`
- For issue 5, use the branch name `COMP23311/EX1/least-magic-gnomes`
- For issue 6, use the branch name `COMP23311/EX1/museum-quest-repeats`

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.

Step 5

Before getting started on the fixes, each team member must ensure that the bug they are responsible for is visible in the test suite. That means that the presence of the bug must be flagged up by at least one failing test. This test can be one that already exists in the code base, or (if no such test exists) you will need to write new tests to make the bug visible.

If the issue is already exposed by the test suite, the responsible team member must make a comment in the issue tracker to say which test or tests are failing because of the presence of the bug. Copy and paste the key parts of the failure report for the tests in question into

the issue tracker comment, and add a sentence or two to explain why you think the failure report is caused by the issue you are fixing.

If the issue is not exposed by the existing test suite, you will need to write at least one test case that fails because of the presence of the bug. You'll need to work out where to put the new test case in the Stendhal code base, so that other developers familiar with the organisation of the code will be able to find it easily. You'll also need to work out how to use the test objects that the Stendhal team have provided, to set up the a game (or partial game) in the state needed to make the issue visible.

You should keep track of how long you spend on this step, to the nearest hour. When the step is completed, add a comment to your issue telling us how long this was. Please use the following phrase, which will be searched for by the automated marking system:

```
Issue now visible in the test suite.
/spend <time spent>
```

where `<time spent>` is replaced with the amount of time you spent on this task, expressed in a form that the GitLab issue time tracking facility can understand.

Step 6

Once you have one or more tests that fail because of the presence of the bug, you can make changes to the production code (the code under the `src` folder) to fix it. This step can be considered complete when the changes you have made in steps 5 and 6 are committed to your feature branch, and the following is true when the feature branch is checked out:

- All the tests you wrote to expose the bug pass.
- Any existing tests you identified as exposing the bug now pass.
- No other tests in the Stendhal test suite fail.

As in step 5, you should keep track of how long you spend on this step, to the nearest hour. When the step is completed, add a comment to your issue recording this. Please use the following phrase, which will be searched for by the automated marking system:

```
Issue resolved.
/spend <time spent>
```

where `<time spent>` is replaced with the amount of time you spent on this task, expressed in a form that the GitLab issue time tracking facility can understand.

You may push your feature branch to your team repository at any point during steps 5 and 6, to make your progress visible to your team and to preserve a record of it on the School GitLab server. You do not have to wait until the issue is completely fixed. Note that we can only mark commits that we can see in your team's remote repository; code changes that exist only in your local repository are invisible to us and won't be marked.

Once you have pushed your feature branch to your team's remote, the continuous integration server will run the automated build and test processes, to determine the health of the code on your branch.

Step 7

The next step is to merge your work with the main development branch. The Stendhal team use the `master` branch as their development branch, and we ask you to continue to follow this convention in your coursework. For this coursework exercise, we ask you to perform merges in your local repository and push them to the team repository. In later exercises, we will make use of Merge Requests on GitLab to help you manage the merging process, but for this exercise you should **not** create any merge requests. Your goal for this coursework exercise is to demonstrate that you understand the basics of merging, by carrying out the steps yourself.

Before merging your work into the development branch on your team's repository, you need to check that your code changes are not going to introduce any unexpected problems. To do this, first, **fetch any changes from your team's repository**, and merge them into your local repository, to make sure you are working from the most up-to-date version of the code base. (If your whole team is following the workflow correctly, this step should be trivial. Advice on how to carry out this fetch process can be found on Blackboard.)

Next, check out the `master` branch and merge your feature branch into it (following the same approach we covered in the GitLab Access Check activity).

Do not push the `master` branch to your team's remote repository at this stage. (You may of course push the feature branches.)

Check that the code base that results from this merge compiles, and that the full test suite passes. If you encounter any problems, you'll need to reset the `master` branch back to the commit it was on before the merge. This will have the effect of undoing the merge you just made.

Fix the problem in your feature branch and start this step again. (Of course, if the problem turns out to be caused by code one of your team members has committed to your team

repository, you'll need to stop development and work with that team member to fix their feature branch and re-merge before coming back to start this step again for your own feature branch.)

Once you are confident you have a clean merge, you can push the changes to the `master` branch to your team repository. (You may wish to fetch commits from your team repository before doing this, if a certain amount of time has elapsed since you started work on this step.)

If you get a clean development branch build from the continuous integration server at this point, you can close the issue in the issue tracker.

Step 8

As in steps 5 and 6, you should keep track of how long you spend on this step, to the nearest hour. When the step is completed, add a comment to your issue telling us this. Please use the following phrase, which will be searched for by the automated marking system:

```
Issue merged into development branch.
/spend <time spent>
```

where `<time spent>` is replaced with the amount of time you spent on this task, expressed in a form that the GitLab issue time tracking facility can understand.

Do not push broken code to your team's development branch.

If, by the deadline, you have a feature branch that contains compile errors or has failing tests, you should push it to the team repository for marking, but you should *not* merge it with your development branch. Your team will get more marks for not merging broken code than you will for allowing broken code to reach the development or release branches.

Leave issues for unmerged feature branches open. The team needs to know that this issue has not yet been fixed, so they can return to it in a future release should customer interest in fixing it become pressing.

Step 9

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

carrying it out. This team member should create an issue for this task called:

```
Prepare release 1.28uom
```

This issue should be associated with the coursework 1 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 issues that have been correctly fixed 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.

Look at the previous releases created by the Stendhal team. You will notice that a number of changes are made in each case. Notably:

- The version number of the software is updated.
- Any new authors are added to the `doc/AUTHORS.txt` file
- A description of the changes included in the release is added to the `doc/CHANGES.txt` file.

You will need to make these changes for your release too. You can add them directly to 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_28_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 final release commit has been pushed.

Step 10

You are done with the technical work at this stage, but there is one more task to do: prepare for the marking interview in week 5. In this interview, your TA will ask you to demonstrate

some of your bug fixes 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 1`.

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 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, and there is no option to submit your work late. 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, and not to be penalised for the work you have not been able to do.

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 ideally be made by the team member recorded as responsible for the commit in the issue tracker.

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.

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.

If a team includes non-contributing members, the marking scheme will be adjusted to take this into account. This means it is not necessary for contributing team members to pick up additional work, to fix issues that have been assigned to non-contributing members. Instead, everyone should concentrate on fixing their own issue, and on including it safely into the release.

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-mail 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.