Week 2 Workshop

Software process models/Intermediate GitLab

This week...

In this week's workshop, you will:

- make sure you understand the process models you've seen in lectures
- explore the relationship between Agile principles and Scrum
- learn how to use branching and merging in your git projects
- see how GitLab's merge requests can be used in a project

Task 1. Decide on a team

You will be doing all of the in-semester work in FIT2101 as part of a team of four students from your lab class. We'll be starting the project soon, so now is the time to pick your team.

Teams may not contain people from more than one workshop as this makes it unclear who is responsible for marking you. If you really want to work with somebody who isn't in your workshop, see if you can transfer so that you are both in the same class. If your target class is full, post to the Moodle forum and ask whether anybody would be willing to swap with you.

Register your team in the student team selection module under Week 2 on Moodle. If you can't decide on a team, we will put you in one; if teams are too small to be viable, we may merge them.

Task 2. Software process models

Your facilitators will put you in breakout rooms for this activity. One person in your breakout must create a Google Doc or Google Sheet and share it with the group. Add to it the following table:

Process model	Iterative?	QA	Risk management
Ad-Hoc			No
Waterfall		Done between phases	
Prototyping	No		
Spiral			
Scrum			Yes; during sprint planning

Using the descriptions in the slides for Lecture 1, and any other reference material you like, fill in this table. Consider whether the resulting table is enough to distinguish between these process models. Do you think any other information about the process models is likely to be important? If so, what?

Task 3. Ad-hoc and Agile

Agile software development is sometimes criticized as being indistinguishable from ad-hoc (or code-and-fix) development because it tends to discourage formal documentation and extensive planning.

Look at the Agile Manifesto in the slides for Lecture 2 or at http://agilemanifesto.org/ and the 12 Agile Principles at http://agilemanifesto.org/principles.html or at https://www.agilealliance.org/agile101/12-principles-behind-the-agile-manifesto/. We've said in lectures that any process model that conforms to these principles can be considered Agile. Do you think ad-hoc development is an Agile methodology? Why or why not? Discuss this with the other students in your breakout[1].

Task 4. Agile principles and practices

The Agile Alliance website has a useful glossary at https://www.agilealliance.org/agile101/agile-glossary/. Using this glossary, and any other resource you like, make sure you understand the following Scrum practices that we saw in Lecture 2:

- Agile teams
- Daily standup meetings
- Sprints (i.e. *timeboxed iterations*)
- Project backlog
- Retrospective

For each practice, describe how it relates to the 12 Agile Principles. Which principles does it support? Are there any principles that conflict with it? Again, discuss this with the other students at your table.

Note: the rest of the Tasks in this workshop require you to use a git repository that is accessible to you and a partner in your class. If you did the git exercises in last week's workshop, you can use the repository you created then; if you skipped it because you're comfortable with basic use of git, or if you deleted the repository you made last week, please create a new repository. Look at the workshop sheet and other resources from Week 1 if you've forgotten how this is done.

We also recommend you re-read the sections on branching, merging, and pull requests in git and GitLab on Alexandria[2] and refer to them as you work.

Task 5. Issues

You'll need a partner or group for this Task. Create or clone a repository on git.infotech.monash.edu. Create a "hello, world" program – i.e. a program that simply outputs the string "hello, world"– commit it, and push it to the server. (It doesn't matter what language you use for this.)

An *issue tracker*[3] is a piece of software that helps developers manage issues – things that they need to do, such as features that need adding, bugs that need to be fixed, or maintenance tasks. GitLab has a built-in issue tracker. You can read about it at https://docs.gitlab.com/ee/user/project/issues/index.html.

Read the GitLab documentation on creating and resolving issues. In your shared repository, have each person create at least one issue. For example, you might want to show Practice creating, viewing, moving, and deleting issues.

Once you have begun work on your project, you may choose to use issue boards to keep track of the work that needs to be done and the tasks assigned to each team member. We will come back to this topic in later weeks.

[1] If you're not in a breakout, ask the host to assign you to one.
[2] https://www.alexandriarepository.org/module/introduction-to-version-control-with-git/
[3] See https://en.wikipedia.org/wiki/Issue_tracking_system

Task 6. Branching and merging in git

Create a new branch using the git client of your choice. Call this branch `hello_me`. In this branch, modify your original program so that, instead of saying "hello, world", it says "hello, *your_name*". Commit this change and push it. Verify that you can see two branches if you view the repository on `github.com`.

Check out your `master` branch again and make sure that the original "hello, world" message is still there. Now *merge* the changes in the `hello_me` branch into `master`.

Task 7. Resolving merge conflicts

In this Task, you'll create a merge conflict and then resolve it. You may work with a partner if you like, with one person working in `master` and one person working in a different branch.

To create a merge conflict, you should begin with a repository containing a "hello, world" program, and create a branch with a modified version of your program in the same way as in the previous Task. This time, however, you should modify the version of your program in `master` – perhaps to make it say "goodbye, world!". Make this change and commit and push it to `master` *after* you create your branch but *before* you merge. This time, when you attempt to merge your branch into `master`, the merge should fail.

If you don't get a conflict when you try to merge, make sure:

- you modify `master` after you've branched
- you modify `master` before you've merged your branch
- you've changed the *same line* in `master` as you've changed in your branch

Examine your conflicted file and compare it to the example on Alexandria. Edit it to resolve the conflict: you want the program to say "hello, *your_name*" rather than "goodbye, world". Ensure that your `master` branch on GitHub now looks the way you want it to.

Task 8. Merge requests

You'll need to work with a partner (or group) for this Task.

Read GitLab's online documentation at https://docs.gitlab.com/ee/user/project/merge_requests/index.html. Using your repositories from the previous two Tasks, or any other repositories you own, have everybody create a branch for themselves, make and commit a small change – ideally, one that won't conflict with anybody else's changes. For example, you could have each person create a new file with a different name[4]. and add it to their branch. Once each person has finished doing this, have them issue a merge request.

When the merge requests have been created, have the owner of the repository go to `github.com` and view the pull requests. Look at each one and decide whether it should be merged into the master branch; if so, merge it. If you get stuck, ask a facilitator for assistance – but do make sure you've read the documentation first! If you like, you can repeat this exercise to make sure each person gets to try managing merge requests, but if you have a large group it's fine if some people just watch what the owner does. As long as you're all confident that you'll know what to do about *real* merge requests, that's enough.

Once you're reasonably sure you understand what's going on, try dealing with merge requests that do conflict.

[4] You won't be able to easily merge two branches that have files with the same name but different contents.