Victoria University of Wellington
School of Engineering and Computer Science

SWEN222: Software Design

Lab Handout

1 Outline

The purpose of this lab is twofold. Firstly, to introduce you to the Git version control system. Secondly, to get some more practice developing GUI programs in Java.

1.1 Life without Version Control

To understand the value of a version control system, consider two developers (Sally and Mark) working on a project (the Simple Lisp Interpreter) *without using a version control system.* Both Sally and Mark begin with identical copies of the source code (i.e. they both download the same `simplelisp.jar` file). Now, let's say Mark is working on the Interpreter package, and makes several improvements to the Java files there. At the same time, Sally is working on the Editor package and spends a lot of time improving the Graphical User Interface (i.e. the `InterpreterFrame.java` file).

At this point, Sally and Mark each have different versions of the Simple Lisp code; they want to merge them back together so they can see the improvements made by each other. The only way to do this is for Sally and Mark to send the files they have changed to the other person. This means they need to record exactly what changes they make when they make them. Otherwise, they may forget get to send an updated file to the other person.

There are several problems with this manual approach to version control:

1. A lot of time is spent recording exactly what changes a person makes.
2. It relies on the diligence of each programmer to record exactly what they have done. Forgetting a change can be catastrophic. Suppose Sally's part of the code uses some method in Mark's part of the code, but Mark forgets to tell Sally he's renamed the method. When Sally integrates Mark's changes, her version of the program will no longer compile. She might waste countless hours trying to figure out where she made a mistake, only to finally realise that Mark has made a change and told her.
3. If either Sally or Mark make some change and, later, decide that they want to undo this change, then this is very difficult unless they have kept regular backups of the integrated project. They need to hunt through these backups to find the unchanged version. Then, they need to integrate that back into the latest version. All of these things take time and, again, rely on programmer diligence to make regular back ups.

4. Suppose Sally finds a bug in Mark's code. Mark then hunts through his back ups to find the point at which the bug appeared. If he finds it, he might simply undo whatever change introduced the bug. What if the change he made was important for some reason? If he can remember why he made that change, then he'll know the answer. But, what if he made that change several months or years ago? If he logs every change made in a ChangeLog file, then he will be able to look back at why he made that change. Again, he needs to be diligent in recording every change in the log.

Each of the above points relies on the programmer to be diligent in what they do. If they forget something (which is easy to do when deadlines have to be met), then problems can arise later on in the project. Version control simply makes these things easier to do properly by automating many parts of the process and by making it easy to look back over what was done. Now, let's see how Sally and Mark could have used version control instead.

1.2 Life with Version Control

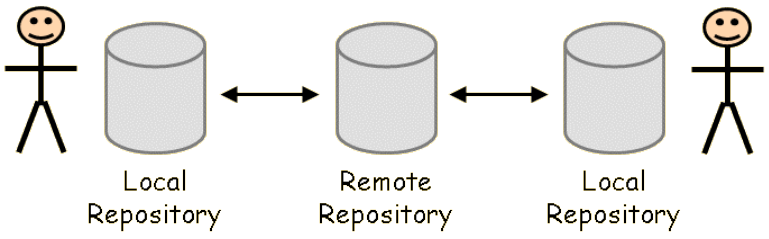
A *version control system* is primarily used to manage and co-ordinate software projects that are being developed by a team of programmers. The system helps to manage the files being developed by all developers. We can think of each file as being given a version number and that, whenever a developer changes a project file, this version number is increased. This way, the system always keeps track of the latest version of the project.

Sally and Mark would begin by creating a shared repository for their project. This is the central place where all the files of the project are stored. When either Sally or Mark has made a change to some file they must **commit** that change to the repository. This means the system will upload the new file into the repository and increase its version number. However, the original version of the changed file is not deleted; instead, it is simply stored under the old version number. This means Sally or Mark can look back through the **history** and see the state of each file before and after every change made to the project.

Before the system commits a file to the repository, it will ask for a **log message** from the person making the change. This is attached to the new file, so that the log entry of every change can be seen. Every so often, Sally and Mark **update** their local version of the project. This means any changes made by the other person will be downloaded from the repository onto their machine, so they are always working on the latest version of the project.

1.3 Version Control with Git

Git is a modern, distributed version control system developed initially by Linus Torvalds. A typical setup of Git is the following:



Here we have just two users, though in practice there could be many more. Each user has a *local* copy of the Git repository (called a *clone*), and there is a shared copy of the repository (called a *remote*) located somewhere (e.g. on GitHub). Users make *commits* to their local repository, and *push* their changes up to the remote repository for others to access. They also *pull* changes made by others into their local repository.

Activity 1 — Git Local Repository (30 mins)

In this activity, you will create a *local* git repository and perform some routine operations on it (e.g. committing code, etc). The instructions are given for performing operations both on the command-line and from within Eclipse. *Dave strongly recommend learning to use Git from the command-line, as this offers the most flexibility and control.*

NOTE: for more info on using Git in Eclipse, see: `https://wiki.eclipse.org/EGit/User_Guide`

Create a new project in Eclipse. For the purposes of this lab, you should create a new repository in Eclipse and add at least one Java file to it. If you wish, you may use an existing project that you don't need any more, or you can copy files over from an existing project into a new project.

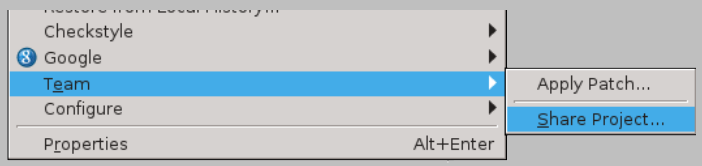
Initialise Git Project. Before we can start using Git, we need to initialise it for our project. This creates a special ".git" subdirectory in our project.

TERMINAL To do this, run the following commands from the terminal:

```
% cd /your/project/directory
% git init
% git checkout -b master
```

This initialises an empty repository in the given directory, and creates a new branch called `master`.

ECLIPSE There is no way to initialise an empty repository in Eclipse. Having run the above commands, you must signal this to Eclipse by selecting "Share Project" from the "Team" menu.



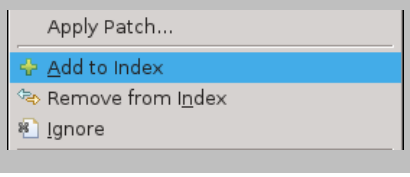
Add file to the index. At the moment, Git does not know anything about the files in your project. To tell Git which files to look after, we need to add them to the *index*.

TERMINAL To do this, run the following command from the terminal:

```
% git add path/to/your/file.java
```

You can check that this file has been added by running `git status` from the command line. You should see your file listed as a "new file", and there maybe other files which are "untracked". At this point, you may wish to add any other files to the index.

ECLIPSE In Eclipse, you can add a file to the index by right-clicking on it and selecting "Add to Index" from the "Team" menu.

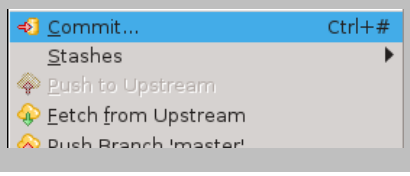


Commit file to repository. We now want to commit our file(s) to the repository, and make an appropriate commit message. A good commit message describes what the commit does.

TERMINAL Run the following command from the terminal:

```
% git commit -m "My commit message"
```

ECLIPSE In Eclipse, you can make a commit by selecting "Commit" from the "Team" menu. You can then just enter the commit message into the text box.



Modify file. At this point, you should make some change to one of the files you have added to the index. For example, renaming a variable or method. You should then commit this change using the same procedure as before.

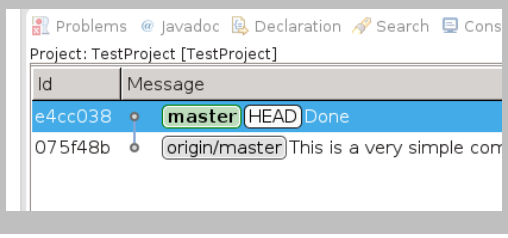
Viewing the Log. We now want to view the log of commits for our project so far. This is useful to remember what we worked on last, and also to explore changes made by others (if we have a shared project).

TERMINAL Run the following command from the terminal:

```
% git log
```

You should now see a short list of commits, including their commit messages.

ECLIPSE In Eclipse, you can view the history by selecting "Show in History" from the "Team" menu.



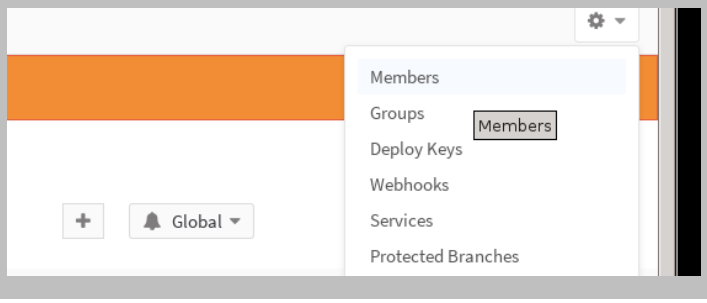
Activity 2 — Git Remote Repository (30 mins)

At this point, you will divide into pairs to work on a shared Git repository.

As a little technical issue, first of all, you both need to logg in ecs gitlab at least once, so that the system recognize your "existence".

Creating the shared repository. The first stage is for one of the pair to create a shared repository on our local GitLab instance.

GitLabs Goto `gitlab.ecs.vuw.ac.nz`, login and create a new repository. Make sure the project is marked as "internal" or "public". Then, edit Settings⟶Members to add your partner as a `Developer` on the repository:



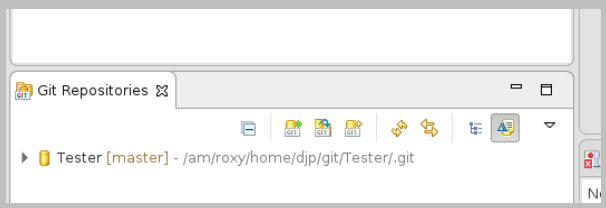
Create Project and Configure Remote. First, we need to clone the remote repository into your Eclipse workspace.

Terminal Run the following command from the terminal:

```
% cd /your/workspace
% git clone https://gitlab.ecs.vuw.ac.nz/path/to/Repo
% cd Repo
% git checkout -b master
```

This should create a new directory in your workspace. Then, create a new project in Eclipse with the same name as your repository.

Eclipse Doing this from Eclipse is difficult (for some reason). The best way is to open the "Git Repositories" view from the Window⟶Open View⟶Other menu. Then create a new repository using the gitlab URI:



At this point, you can create an empty Java project and, via the "Team" menu, share it using the repository you just created.

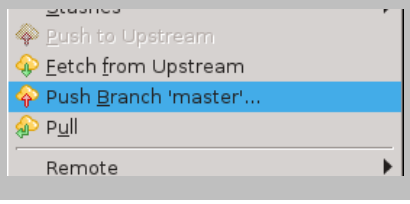
Create and add files. Create some files in your repository and then add them to the index as we did for the local repository before. You can copy over some files from another project if you like.

Push to remote repository. At this point, we want to push all of our files from our local repository into our shared repository.

TERMINAL Run the following command from the terminal:

```
% git push origin master
```

ECLIPSE From the "Team" menu, select "Push branch master". This will then open up a dialog which identifies your shared repository as the "remote".

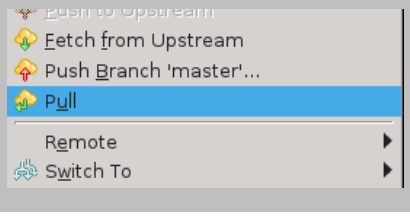


Pull from remote repository. Once both partners have pushed code into the shared repository, each should then pull all of the files into his/her local repository. This will include any files which our partner pushed into the repository as well.

TERMINAL Run the following command from the terminal:

```
% git pull origin master
```

ECLIPSE From the "Team" menu, select "Pull".



2 Real Example: Battleships (60mins)

At this stage, you will now gain some more experience using git on a realistic (but small) code base. In particular, you will implement a *Graphical User Interface* for the well-known battleships game. You should continue to work in pairs and use git throughout the development of the code. In particular, you should get into the habbit of committing your changes frequently, and in providing useful commit messages.

NOTE: you do not need to complete the game in order to do well in this lab; rather, the emphasis is on the use of git — *not completing the coding*. One goal of this coding part is also to expose you to java Swing, that will be used, expanded and explained more in the rest of the course.

2.1 Application Window

To begin, you should download and run the code for the battleships game (found under `LectureSchedule` on the SWEN222 course homepage). Currently, the game provides a text interface for viewing the battle grid and playing the game.

Also provided with the code for the Battleships game is the skeleton of a graphical user interface, provided in the file `swen222.battleships.GraphicalInterface.java`. If you execute this class, you will find that an empty application window is created.

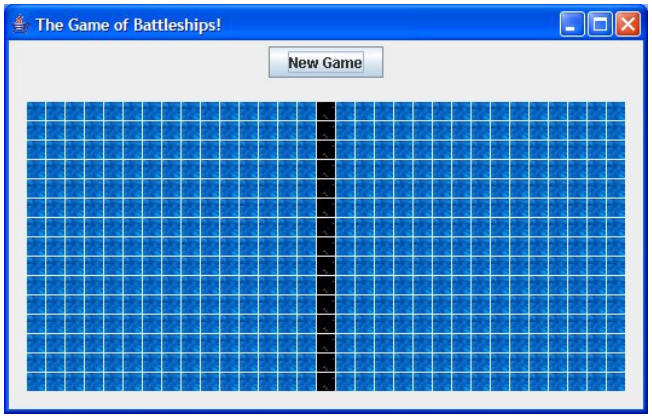
You should now complete the application window so that it draws an empty battle grid, and provides a button for starting a new game. To do this, you need to complete the method `makeOutermostPanel()`. This method must:

1. Construct a `JPanel` for representing the left and right battle grids. This should use a GridLayout which is large enough to contain both the left and right grids, as well as a single partition square in the middle. Each square in the left and right battle grids should be initialised with a JLabel containing the emptySquare icon. The horizontal and vertical gap between grid locations should be set to 1, and an empty border of sufficient size should be placed around the `JPanel`.

2. Construct another `JPanel` for holding the "new game" button, and add that button to the `JPanel`. This should use the FlowLayout to ensure the button is located in the center.

3. Finally, construct the outermost `JPanel`, whilst adding the battle grid `JPanel` and button `JPanel` (put the buttons above the battle grid).

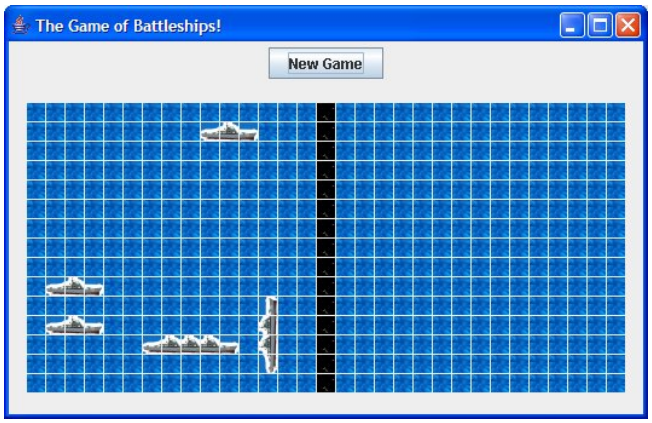
The game window should now look something like this:



2.2 Game Board

The next stage is to draw the game board by completing the method `drawBoard()`. This method must iterate the left and right battle grids setting the icon for the corresponding JLabel object appropriately. You will find a partially implemented method called `getSquareIcon()` which you will need to complete. This method should be used to determine the appropriate icon for a particular square, depending upon whether or not it is in the left or right grid.

With the `drawBoard()` method properly implemented, you should find the game window now looks like this:



2.3 Playing the Game

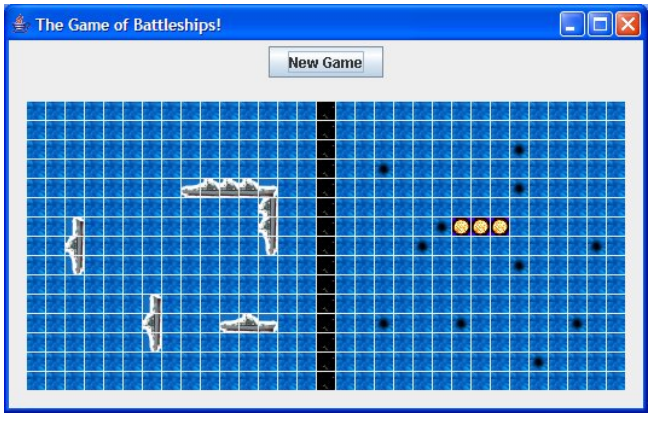
The final component is to allow the user to play the game! To do this you need to:

1. Modify the `GraphicalInterface` class so that it implements `MouseListener`, and add the methods required by this interface (they can be empty for at first).

2. Modify the initialisation of the `JLabel` objects found in the right battle grid to use this object as the `MouseListener` (via method `addMouseListener()`). The reason for doing this is to ensure that mouse click events which are made on the grid `JLabel` objects are redirected to the `GraphicalInterface` object.

3. Implement the method `mouseClicked(MouseEvent e)`. This method should first determine the object that was clicked upon using `getComponent()`; if this was a `JLabel`, it should iterate the right battle grid to identify which square was clicked on. Once the right square is identified, the `BattleShipGame.bombSquare()` method should be called to update the game state. Finally, the method should call `drawBoard()` to update the display.

At this stage, the player should be able to play the game. For example, after a number of bombs have been deployed the game window might look like this:



There are many aspects of the game which are not yet implemented. In particular, the computer is not yet playing the game! If you have time remaining, you should attempt to implement as much extra functionality as possible to make the game playable. In particular, ensure that the game can be restarted via the "new game" button, that the computer is actually playing, that the score is reported in a status bar at the bottom and that a winner is announced which either side has all of its ships destroyed.

Marking Guide

Each lab is worth roughly 1% of your overall mark for SWEN222. The lab should be marked during the lab sessions, according to the following grade scale:

- **0**: Student didn't attend lab.
- **E**: Student did not really participate in the lab.
- **D**: Student's participation was *poor*. For example, he/she did not appear to be involved or interested in examining the other students' designs and/or code.
- **C**: Student's participation was *satisfactory*. That is, he/she some progress on first part of the lab.
- **B**: Student's participation was *good*. That is, he/she made made good progress on first part of the lab.
- **A**: Student's participation was *excellent*. That is, he/she completed first part of the lab and made progress on second part using git; **NOTE:** *emphasis for second part is on use of git, not completing the code*