Agile Development

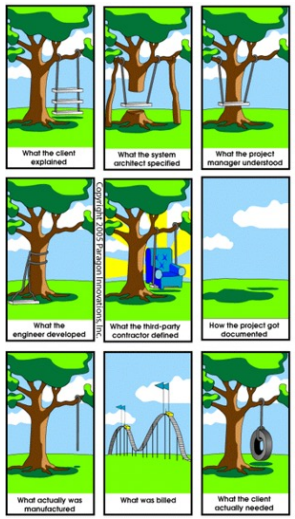
CITS3403 and CITS5505 Agile Web Development

From Agile in a Nutshell, by Jonathan Rassmusson
Further reading: The Agile Handbook

Semester 1, 2024

The Agile methodology

Software development is hard...

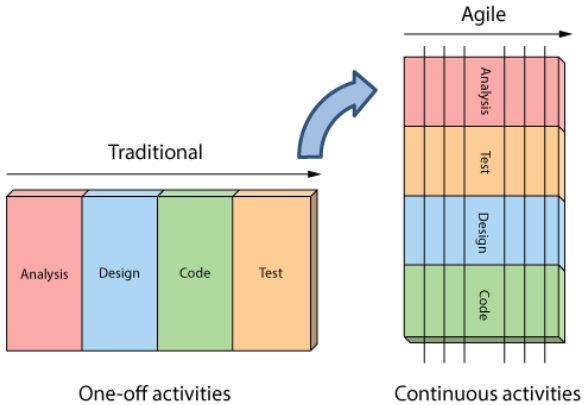


The Waterfall method of project management is strictly sequenced: you don't start design until research is done and you don't start development until the designs are signed off on, etc.

If things go wrong in the waterfall software projects, they tend to go *very* wrong.

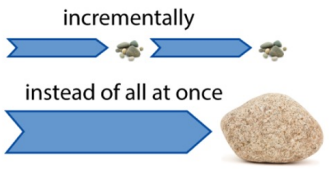
Agile is a way to manage projects that aims to minimise the problems when things go wrong. It can be used for virtually anything, but it was founded in software development.

Agile development is continuous...



The basics of Agile

- "The Agile Handbook" focuses on Agile for software development, but many of the principles can be expanded to other fields.
- Agile breaks down larger projects into small, manageable chunks called iterations.

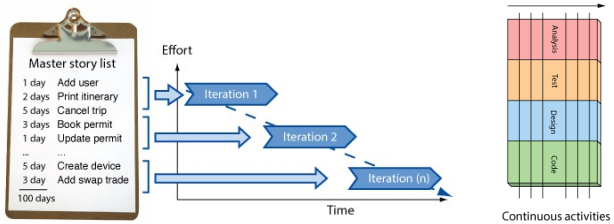


- At the end of each iteration (which generally takes place over a consistent time interval) something of value is produced.
- The product produced during each iteration should be able to be put into the world to gain feedback from users or stakeholders.

How does Agile work?

Within a single iteration there are several phases:

1. sort the current list of desirable features in terms of difficulty and priority
2. start developing the features at the top of the list
3. at the end of the iteration, show what you have to the user
4. update the list of features:
   - removing the features you've completed
   - adding new ones and reprioritise existing ones using the user's feedback…



12 key principles of Agile

1) Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

2) Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

3) Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

4) Businesspeople and developers must work together daily throughout the project.

5) Build projects around motivated individuals. Give them the environment and support they need and trust them to get the job done.

6) The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

12 key principles of Agile

7) Working software is the primary measure of progress.

8) Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

9) Continuous attention to technical excellence and good design enhances agility.

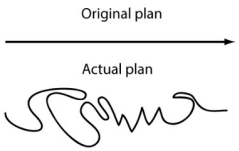
10) Simplicity - the art of maximizing the amount of work not done - is essential.

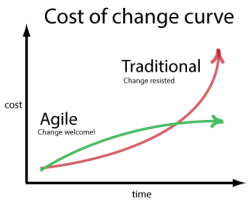
11) The best architectures, requirements, and designs emerge from self-organizing teams.

12) At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.

How is Agile different from Waterfall?

- Development is iterative
- Planning is adaptive
- Roles blur
- Requirements change
- Working software





Myths about Agile

1. Agile is a silver bullet
2. Agile is anti-documentation
3. Agile is anti-planning
4. Agile is undisciplined
5. Agile requires a lot of rework
6. Agile is anti-architecture
7. Agile doesn't scale

Agile approaches: user stories

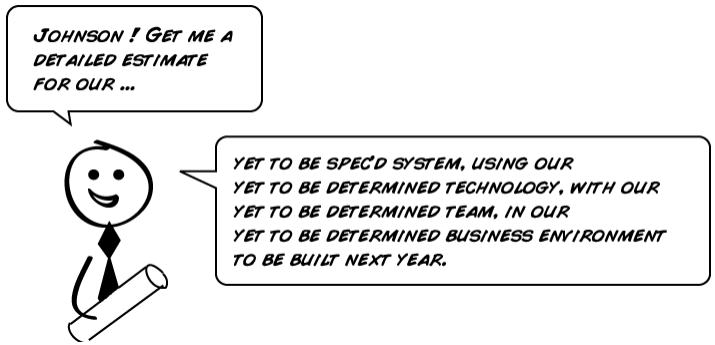
A list of features can be developed by using user stories

- Each user story describes one desirable feature of the software.
- They are told from the end user point of view.
- These features can be delivered in short units of work.
- They are often written on cards to facilitate communication

#	Backlog Item (User Story)	Story Point
1.	As a Teller, I want to be able to find clients by last name, so that I can find their profile faster	4
2.	As a System Admin, I want to be able to configure user settings so that I can control access.	2
3.	As a System Admin, I want to be able to add new users when required, so that...	2
4.	As a data entry clerk, I want the system to automatically check my spelling so that...	1

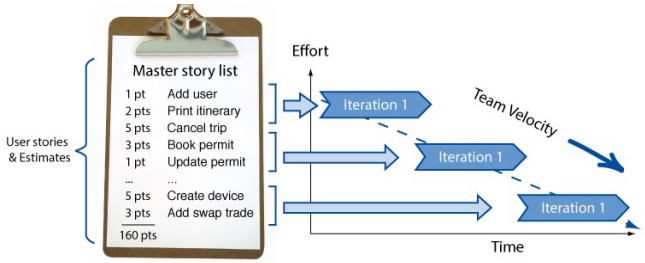
Agile approaches: estimation

- Time estimation for features is difficult but essential.
- You should always practice estimating the amount of time development will take.



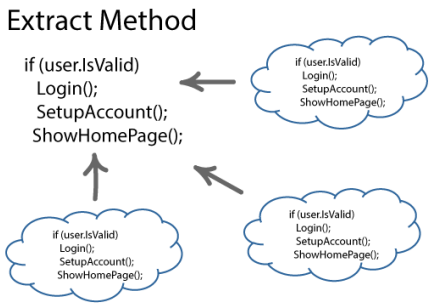
Agile approaches: planning

- Combines the user stories and estimations to build a feasible plan for delivery.



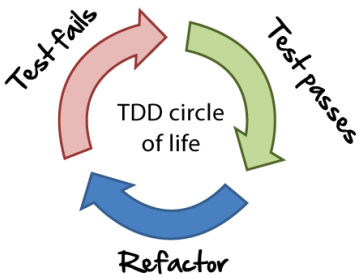
Agile approaches: refactoring

- To maintain a design and functionality, we must be prepared to refactor code.
- Organise code into manageable modules.
- Don't repeat yourself (DRY)



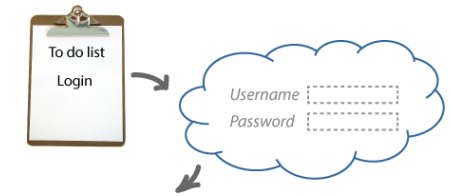
Agile approaches: test-driven development

- Write tests at the start and then write code to pass the tests.
- The tests become the de facto documentation for the system.



Agile approaches: unit testing

Unit tests are snippets of test code developers write to prove to themselves that what they are developing actually works. Think of them as codified requirements.

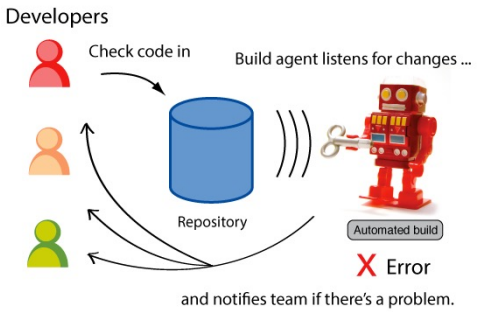


Assert.IsValid("username", "password");
Assert.IsNotValid("username", "invalid password");

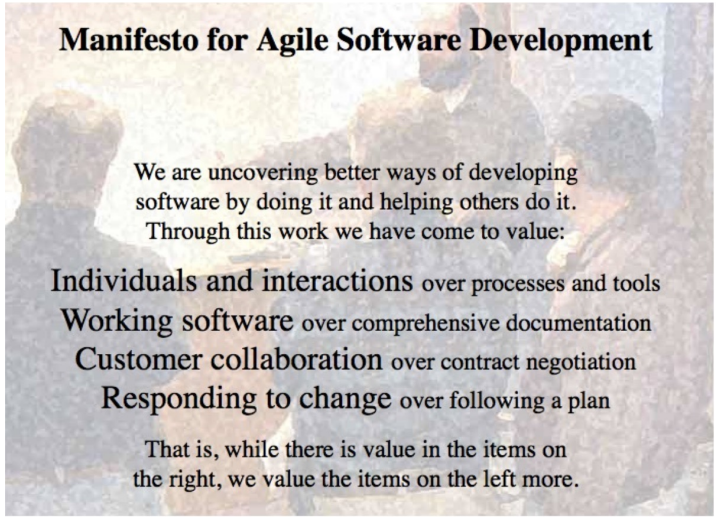
They are powerful because when combined with a continuous integration process they enable us to make changes to our software with confidence.

Agile approaches: continuous integration

- Continuous integration keeps the code in a repository that is automatically maintained, and everyone works on at the same time.



The Agile manifesto



Flavours of Agile

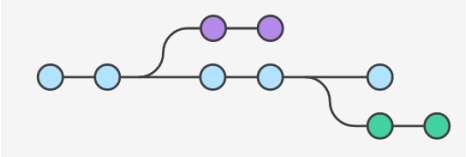
- There are many different flavours of Agile. A non-exhaustive list:

1. Scrum - short, iterative cycles (called sprints) in which the team pulls from a list of requirements that have been prioritized so that the features developed first are of the highest value to customers
2. Kanban - another pull-based system where work is visualized across all team members using a kanban board. Aims to limit work-in-progress and eliminates over-production.
3. Extreme Programming (XP) - an approach intended to improve software quality and responsiveness to changing customer requirements through frequent releases in short development cycles aided by automatic tests.
4. Lean software development, Crystal etc.

Version control: Git

Version control systems

- Writing large pieces of software is hard!
- Many people working on the same code base at once and one frequently makes mistakes that you may only discover weeks or months later.



- We want the ability to resolve conflicts between people and to be able to rollback work.
- Software to do this is called a version control system.
- Many variants out there: Git, Subversion, Mercurial etc.

What is Git?

- Git is a distributed version control system developed in 2005 by Linus Torvalds
- Now the most widely used version control system in the world.

- Git can manage different branches of a development, allowing teams to work on the latest branch, roll back changes, or develop features independently.
- This ability to collaborate on a single code-base while working on different features synergises well with Agile development.

Setting up a Git project

- An individual Git project is developed in a Git repository.
- You can initialise a new repository in the current folder:

```
git init
```

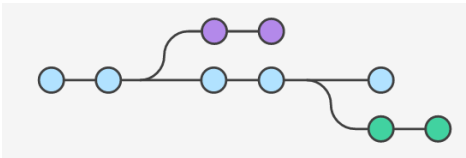
- You can clone an existing repository from elsewhere, makes a new local copy of all files and history of the repository you're cloning:

```
git clone usr@url:path
```

- Theoretically Git projects are fully decentralised - a newly cloned repository is indistinguishable from the copy of the repository it was a copied from.
- As we will see, from a project management point of view, this tends not to be true. There is usually one copy of the repository that is viewed as the "true" state of the project.

Git project structure

- The history of the project is structured as a graph (is not linear!).
- Each node represents one set of changes to the code and are called commits.



- Each commit has a unique hash associated with it known as the commit ID, e.g.

  d259f0481b887be41efd339bb04dc05a6023fa7b

- A set of sequential commits is known as a branch.
- The current branch that you are on is known as HEAD.

Adding new commits

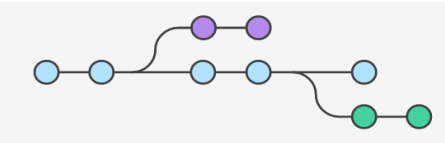
- To add a new commit to HEAD (i.e. the current branch) you can:
  1. (Optional) See the list of files changed since the last commit
     `git status`
  2. Add the set of changed files you want to include in the commit:
     `git add <filename>`
  3. Finalise the changes committed to a branch.
     `git commit -m "msg"`
- **Note:** running these commands only adds the new commit to your repository. It will not make that commit available to other copies of the repository (see subsequent slides).
- Each commit can be undone and replayed.

The art of writing commit messages

- When your commit invariable introduces a bug, or people are trying to work out why you made the changes you did, they will look at your commit message.
- Good commit messages should not only summarise *what* was changed but also *why* it was changed.
- For example:
  Bad: Updated login page
  Okay-ish: Updated margins on login box
  Good: Updated margins on login box to prevent it overlapping with logo.
- Writing good commit messages is an art form!

Managing branches

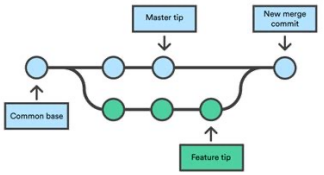
- To create a new branch from your current place in the tree:
  `git checkout -b <branchName>`



- To move to a different branch in the tree:
  `git checkout <branchName>`
- To delete a branch from your local copy:
  `git branch -d <branchName>`
- To see the differences between two branches:
  `git diff <b1> <b2>`

Merging branches

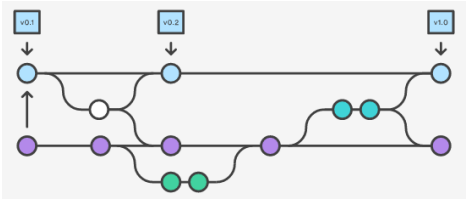
- Not much point to branching forever and ending up with more and more versions.
- Branches can be combined by merging one branch into another.
  `git merge <branchName>`
- Each project has a main/master branch which is considered the "current state" of the project.



- New features are developed on a branch, and then, once complete, merged them back into the main branch.

Git tags

- Sometimes an individual commit is important, e.g. represents a public release, or the first working version with a particular feature.



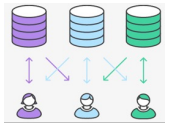
- In that case you can label the commit with a name called a tag.

```
git tag <tagname> <commitId>
```

- Tagged commits can be checked out just like branches

```
git checkout <tagname>
```

Remote repositories



- Your graph is not the same as the graph that another person has!
- You need to be able to synchronise your history with other people.
- Other copies of the repository that you interact with are known as remotes.
- By default, when you clone a repository, a new remote is added with the name origin.
- You can add a new remote to your system with the command:

```
git remote add <remoteName> <remoteURL>
```

- For example:

```
git remote add group1 https://github.com/User1/AgileWeb
```

Other people repos

- You can push your version of the current branch to another remote.

```
git push <remoteName> <branchName>
```

- Equally, you can pull someone else's version of the current branch from their repo.

```
git pull <remoteName>
```

- A standard workflow is:

```
git checkout main
git pull origin
git checkout -b newFeature
…
git commit -m "msg"
git merge main
git push origin main
```

Merge conflicts

- Different features on different branches may require touching the same code.

Original

```
function run(x) {
  return x + 2;
}
```

Branch 1

```
function add(x) {
  return x + 2;
}
```

Branch 2

```
function run(a) {
  return a + 2;
}
```

- Therefore, when merging two branches you may get a merge conflict.

```
<<<<<<< HEAD (Current Change)
function add(x) {
  return x + 2;
=======
function run(a) {
  return a + 2;
>>>>>>> feature2 (Incoming Change)
}
```

manually resolves to

```
function add(a) {
  return a + 2;
}
```

- To resolve the merge conflict, you must manually fix the conflicts and then add your changes for each conflicted file back to the merge commit:

```
git add <filename>
```

- After having done this for all files, complete the merge by running:

```
git merge --continue
```

Avoiding merge conflicts

- Merge conflicts can get really bad!

- They provide a *strong incentive* to keep commits appropriately sized, (i.e. limited to a single feature or bug fix) and to merge often.
- Git only detects syntactic conflicts where changes affect the same lines of code.
- Git does not detect semantic conflicts, e.g. one branch changes how a function works, while another branch calls it relying on the old behaviour. For that you need tests!

Git utilities

- Get a list of all commits and commit messages on the current branch:

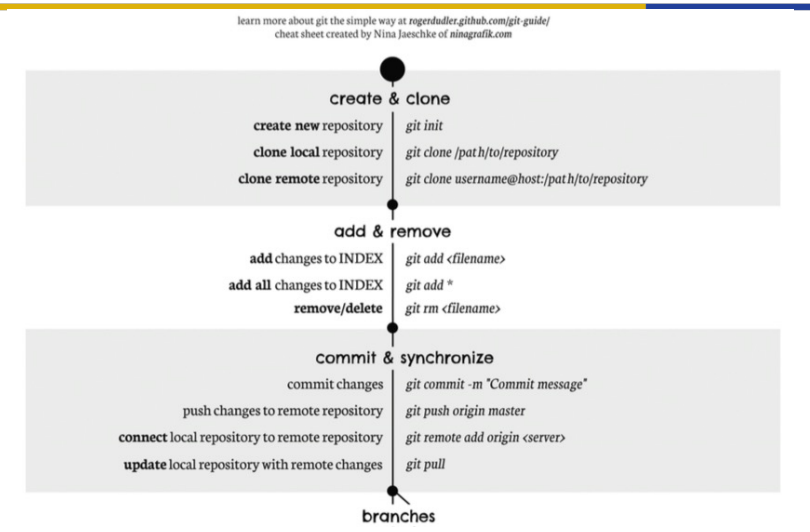
```
git log
```

- Rollback uncommitted changes to a single file:

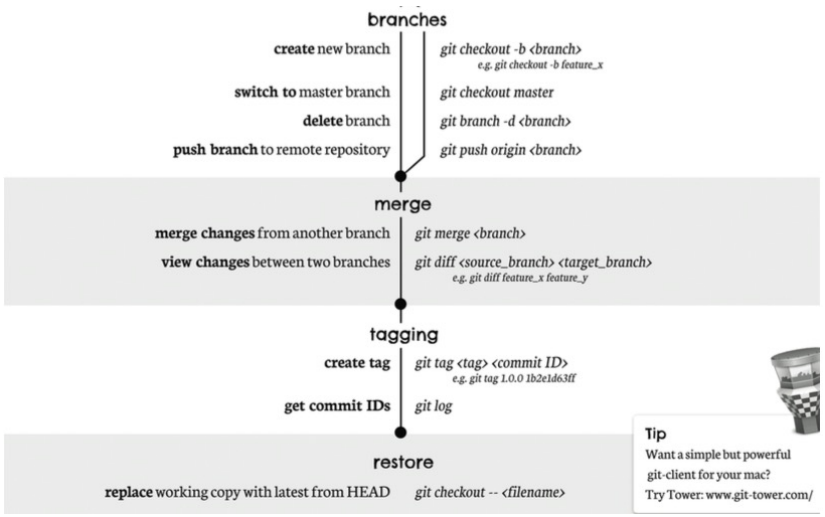
```
git checkout <fileName>
```

- Most modern IDEs have GUI interfaces to Git to avoid you having to type commands manually (e.g. VSCode)

Git cheat sheet



Git cheat sheet



GitHub

What is GitHub?

- GitHub is a service that hosts Git repositories (repos).

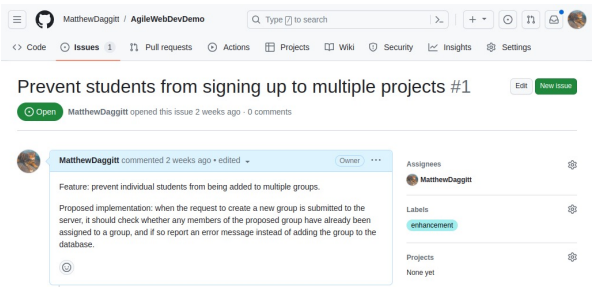
- You can develop collaboratively and use GitHub as the "main" copy of the repo.
- GitHub repos can be public or private. If they are public, anyone can see your code.
- Many other similar services out there:
  - e.g. Bitbucket from Atlassian

Agile project management in GitHub

- GitHub is a great way to manage an Agile project:
  - Everyone works by pushing and pull from the GitHub repo to their local copy.
  - Use the Issues tab to track and discuss your feature list.
  - Use the Pull Requests tab to review feature branches before merging.
- Other features not required to be used in this course:
  - Use Releases to signify a new version of the software.
  - Use GitHub workflows and the Actions tab to automatically:
    - run your test suite before merging a branch in.
    - deploy new releases of your software.

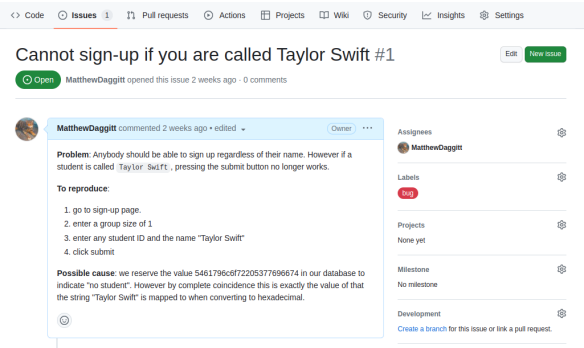
Issues – new features

- Before developing a new feature, you should create an Issue to plan and document it. The first post in the issue should contain:
  - What its purpose is (e.g. to allow the user to do Z)
  - A proposal for how it should be done (e.g. add field A to the database, add widget B to the screen C).



Issues – bug reports

- When you come across a bug, you can create an Issue to document it. The first post in the Issue should contain:
  - Description of both the expected behaviour and the actual behaviour.
  - Description of how to recreate the bug so other people can verify it.
  - Any hypotheses or cause (if known) why the bug might be happening.



Issues – generally

- Issues should contain enough detail that anybody from your team could fix the problem! This is important if one of your team members suddenly falls ill or leaves the course.

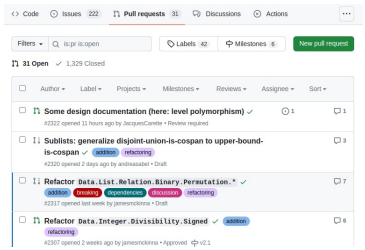
- Issues also serve as forums for discussion. Other members of the team can add their comments below, e.g.
  - How best to fix the bug.
  - Discuss alternative designs for a feature.

Pull requests

- Previously, the workflow was to merge your branch into the `main` locally and then push.

- A much better way is to instead the push the branch to the GitHub repository without merging into main and then open a Pull Request.

- This allows others to look at your code and provide a code review, making suggestions about how to improve it.



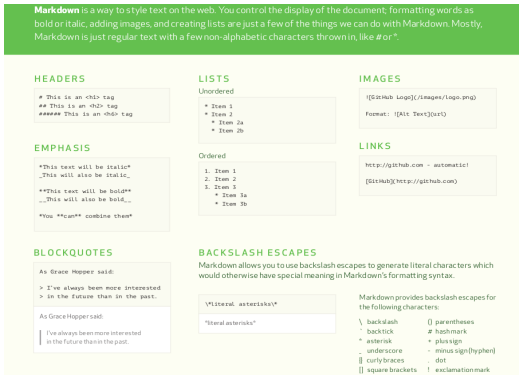
Pull requests

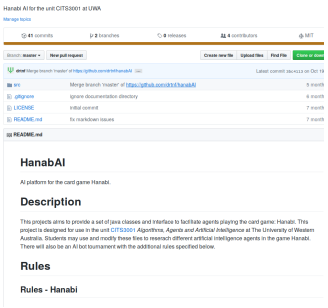
- On an open pull request, GitHub shows all the changes you've made, and other people can add comments, make suggestions etc.
- You can respond to people's comments and update your PR by adding new commits to your branch and pushing.

- When everyone is satisfied you can hit the "Merge" button at the bottom of the PR.

README.md files

- By convention, GitHub repositories have a README.md file written in markdown (a lightweight subset of HTML) that contains instructions about how to build/run your project).
- This file is automatically rendered by GitHub on the landing page for the repository.





The Group Project

An Agile group project

- You will organise your group project via a private GitHub repository, which will form part of the submission of the project.
- Marks will be given for evidence of Agile development practices, i.e.
  - appropriately sized commits with meaningful messages.
  - planning of short-term goals via the Issues tab.
  - reproducible bug reports via the Issues tab.
  - use pull requests to merge in new features via the Pull Requests tab.
  - exhibiting teamwork by contributing to discussions on the Issues
  - exhibiting teamwork by adding code reviews to other people's Pull Requests.
  - intermediate deliverables, pinpointed with Git tags.
- Working together in-person is of course permissible and encouraged, but any decisions made about future work should be entered as new Issues for documentation purposes.
- Some marks will be assigned individually, looking at evidence from GitHub. There will also be an opportunity during the demonstrations to discuss who did what, and what the challenges were, and marks among the group may be split accordingly.

Intermediate deliverables

- You are acting as both developers and users so no opportunity to get iterative feedback.
- The only requirement is that deliverables should showcase some major new piece of functionality that a hypothetical user could give feedback on.
- Given the structure of the course, a sensible first deliverable would be static HTML and CSS mock-ups of your main pages for your website.
- However, it's ultimately up to you to decide what are suitable intermediate deliverables for your project!

An Agile group project

Some advice:

1. Manage expectations. It's a better strategy to set incremental goals for each group member, rather than expect everyone to suddenly become elite coders.

2. Break the project up into small units. Create and assign GitHub *Issue*s that can be worked on individually, and make sure that you deliver your allocation, and then try to help others with theirs.

3. Document your progress. If you run into problems, document and discuss it on the accompanying GitHub *Issue*. If things fall apart, this will salvage your marks at least.

4. Share your knowledge. It's not your responsibility to teach others, but it is part of the Agile methodology. Your team are a valuable resource that needs to be optimally leveraged. Find achievable tasks for everyone, and a short instructional session early can be a big benefit later.