CS2103T — Week 2 — Topics

W2.1 — SE: Intro

W2.2 — SDLC Process Models: Basics

W2.3 — RCS: Revision History

W2.4 — RCS: Remote Repos

W2.5 — IDEs: Basic Features

W2.6 — Automated Testing of Text UIs

W2.1 — SE: Intro

W2.1a - Software Engineering - Introduction - Pros and cons

Pros:

1. Sheer joy of making things
2. Pleasure of making things that are useful to others
3. Fascination of fashioning complex puzzle-like objects of interlocking moving parts and watching them work in subtle cycles
4. Joy of always learning
5. Delight in working in such a tractable (easy to control / deal with) medium

Cons:

1. One must perform perfectly
2. Dependence on others (objectives, resources, information)
3. Debugging sucks
4. Debugging leads to more debugging (linear convergence)
5. Product appears to be obsolete upon / before completion

W2.2 — SDLC Process Models : Basics

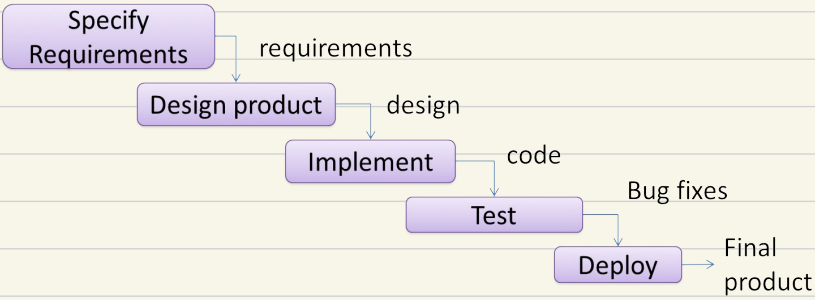
W2.2a — Project Management — SDLC Process Models - Introduction — What

- Software development life cycle (SDLC)
- Several approaches (aka SDLC models / software process models)

W2.2b — Project Management — SDLC Process Models - Introduction — Sequential models

- aka waterfall model
- models SD as a linear process

- harder to implement in practice



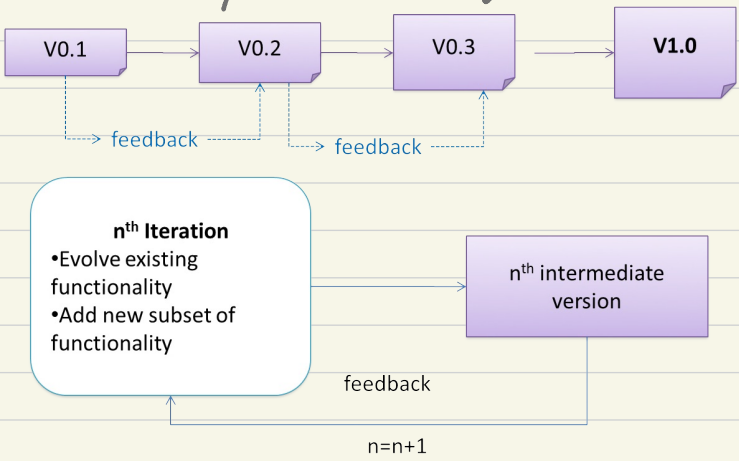
When one stage of the process is completed, it should produce some artifacts to be used in the next stage. For example, upon completion of the requirement stage a comprehensive list of requirements is produced that will see no further modifications. A strict application of the sequential model would require each stage to be completed before starting the next.

This could be a useful model when the problem statement that is well-understood and stable. In such cases, using the sequential model should result in a timely and systematic development effort, provided that all goes well. As each stage has a well-defined outcome, the progress of the project can be tracked with a relative ease.

The major problem with this model is that requirements of a real-world project are rarely well-understood at the beginning and keep changing over time. One reason for this is that users are generally not aware of how a software application can be used without prior experience in using a similar application.

W2.2b - Project Management - SDLC Process Models - Introduction - Iterative models

- aka iterative and incremental
- Similar to several cycles of the sequential model



In this model, each of the iterations produces a new version of the product. Feedback on the version can then be fed to the next iteration. Taking the Minesweeper game as an example, the iterative model will deliver a fully playable version from the early iterations. However, the first iteration will have primitive functionality, for example, a clumsy text based UI, fixed board size, limited randomization etc. These functionalities will then be improved in later releases.

The iterative model can take a *breadth-first* or a *depth-first* approach to iteration planning.

- **breadth-first**: an iteration evolves all major components in parallel e.g., add a new feature fully, or enhance an existing feature.
- **depth-first**: an iteration focuses on fleshing out only some components e.g., update the backend to support a new feature that will be added in a future iteration.

Most project use a mixture of breadth-first and depth-first iterations i.e., an iteration can contain some breadth-first work as well as some depth-first work.

W2.3 — RCS: Revision History

W2.3a — Project Management - Revision Control - What

- Revision control (RC): process of managing multiple versions of a piece of information
- Revision: a state of a piece of information at a specific time that is a result of some changes to it
- Revision control software (RCS): software tools that automate the process of RC
  - will track the history and evolution of your project
  - makes it easier for you to collaborate
  - can help you recover from mistakes
  - will help you work simultaneously on, and manage the drift between, multiple versions of your project (branching)
- aka Version Control Software (VCS)

W2.3b — Project Management - Revision Control - Repositories

- Repository (repo for short): the database of the history of a directory being tracked by an RCS software
- Stores meta-data about revision history
- e.g. of RCS: Git
- Git uses a hidden folder named .git inside working directory

W2.3c — Git and GitHub - init: Getting started

W2.3d - Project Management - Revision Control - Saving history

- Tracking and ignoring:

In a repo, you can specify which files to track and which files to ignore. Some files such as temporary log files created during the build/test process should not be revision-controlled.

- Staging and committing:

Committing **saves a snapshot of the current state of the tracked files in the revision control history. Such a snapshot is also called a *commit* (i.e. the noun).**

When ready to commit, you first *stage* the specific changes you want to commit. This intermediate step allows you to commit only some changes while saving other changes for a later commit.

- Golden rule of RC: "Only commit related changes!"
  i.e. 1 commit = 1 topic

W2.3e - Tools - Git and GitHub - commit: Saving changes to history

- Working directory: the root directory revision-controlled by Git (e.g. the directory in which the repo was initialized)
- Commit: noun: a change (aka a revision) saved in the Git revision history
  verb: the act of creating a commit, i.e. saving a change in the working directory into the Git revision history
- Stage: verb: instructing Git to prepare a file for committing
  ↳ aka add

W2.3f - Tools - Git and GitHub - Omitting files from revision control

- .ignore, .gitignore

The `.gitignore` file tells Git which files to ignore when tracking revision history. That file itself can be either revision controlled or ignored.

- To version control it (the more common choice – which allows you to track how the `.gitignore` file changed over time), simply commit it as you would commit any other file.
- To ignore it, follow the same steps you followed above when you set Git to ignore the `temp.txt` file.
- It supports file patterns e.g., adding `temp/*.tmp` to the `.gitignore` file prevents Git from tracking any `.tmp` file in the `temp` directory.

W2.3g - Project Management - Revision Control - Using history

RCS tools store the history of the working directory as a series of commits. This means you should commit after each change that you want the RCS to 'remember' for us.

Each commit in a repo is a recorded point in the history of the project that is uniquely identified by an auto-generated hash e.g. `a16043703f28e5b3dab95915f5c5e5bf4fdc5fc1`.

You can tag a specific commit with a more easily identifiable name e.g. `v1.0.2`

To see what changed between two points of the history, you can ask the RCS tool to *diff* the two commits in concern.

To restore the state of the working directory at a point in the past, you can *checkout* the commit in concern. i.e., you can traverse the history of the working directory simply by checking out the commits you are interested in.

W2.3h - Tools - Git and GitHub - tag : Naming commits

Each Git commit is uniquely identified by a hash e.g., `d670460b4b4aece5915caf5c68d12f560a9fe3e4`. As you can imagine, using such an identifier is not very convenient for our day-to-day use. As a solution, Git allows adding a more human-readable *tag* to a commit e.g., `v1.0-beta`.

W2.3i - Tools - Git and GitHub - diff : Comparing revisions

W2.3j - Tools - Git and GitHub - checkout : Retrieving a specific revision

W2.3k - Tools - Git and GitHub - stash : Shelving changes temporarily

You can use the git's *stash* feature to temporarily shelve (or stash) changes you've made to your working copy so that you can work on something else, and then come back and re-apply the stashed changes later

W2.4 — RCS : Remote Repos

W2.4a — Project Management - Revision Control - Remote repositories

Remote repositories are repos that are hosted on remote computers and allows remote access. They are especially useful for sharing the revision history of a codebase among team members of a multi-person project. They can also serve as a remote backup of your code base.

It is possible to set up your own remote repo on a server, but the easier option is to use a remote repo hosting service such as GitHub or BitBucket.

You can *clone* a repo to create a copy of that repo on another location in your computer. The copy will have even the revision history of the original repo i.e., identical to the original repo. For example, you can clone a remote repo onto your computer to create a local copy of the remote repo.

When you clone from a repo, the original repo is commonly referred to as the *upstream* repo. A repo can have multiple upstream repos. For example, let's say a repo `repo1` was cloned as `repo2` which was then cloned as `repo3` In this case `repo1` and `repo2` are upstream repos of `repo3` .

You can *pull* from a repo to another, to receive new commits in the second repo, if the repos have a shared history. Let's say some new commits were added to the upstream repo after you cloned it and you would like to copy over those new commits to your own clone i.e., sync your clone with the upstream repo. In that case you pull from the upstream repo to your clone.

You can *push* new commits in one repo to another repo which will copy the new commits onto the destination repo. Note that pushing to a repo requires you to have write-access to it. Furthermore, you can push between repos only if those repos have a shared history among them (i.e, one was created by copying the other at some point in the past).

Cloning, pushing, and pulling can be done between two local repos too, although it is more common for them to involve a remote repo.

A repo can work with any number of other repositories as long as they have a shared history e.g., `repo1` can pull from (or push to) `repo2` and `repo3` if they have a shared history between them.

A *fork* is a remote copy of a remote repo. As you know, cloning creates a local copy of a repo. In contrast, Forking creates a remote copy of a Git repo hosted on GitHub. This is particularly useful if you want to play around with a GitHub repo but you don't have write permissions to it; you can simply fork the repo and do whatever you want with the fork as you are the owner of the fork.

A *pull request* (PR for short) is mechanism for contributing code to a remote repo. i.e., "I'm *requesting* you to *pull* my proposed changes to your repo". For this to work, the two repos must have a shared history. The most common case is sending PRs from a fork to its upstream repo.

W2.4b - Tools - Git and GitHub - clone: Copying a repo

- Remote repo —clone→ local repo
- Remote repo —fork→ remote repo

W2.4c - Tools - Git and GitHub - pull, fetch: Downloading data from other repos

- origin: name of repo we cloned from
- other repo —fetch→ —merge→ your repo

W2.4d - Tools - Git and GitHub - fork: Creating a remote copy

W2.4e - Tools - Git and GitHub - push: Uploading data to other repos

You can push to repos other than the one you cloned from, as long as the target repo and your repo have a shared history.

You can even push an entire local repository to GitHub, to form an entirely new remote repository. For example, you created a local repo and worked with it for a while but now you want to upload it onto GitHub (as a backup or to share it with others). The steps are given below.

W2.5 – IDEs: Basic Features

W2.5a – Implementation – IDEs – What

- Integrated Development Environments (IDEs)

An IDE generally consists of:

- A source code editor that includes features such as syntax coloring, auto-completion, easy code navigation, error highlighting, and code-snippet generation.
- A compiler and/or an interpreter (together with other build automation support) that facilitates the compilation/linking/running/deployment of a program.
- A debugger that allows the developer to execute the program one step at a time to observe the run-time behavior in order to locate bugs.
- Other tools that aid various aspects of coding e.g. support for automated testing, drag-and-drop construction of UI components, version management support, simulation of the target runtime platform, and modeling support.

Examples of popular IDEs:

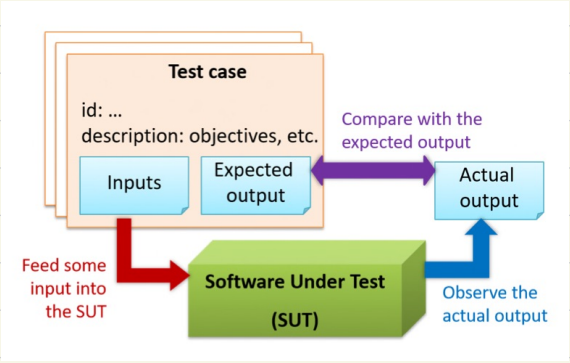
- Java: Eclipse, Intellij IDEA, NetBeans
- C#, C++: Visual Studio
- Swift: XCode
- Python: PyCharm

W2.5b – Tools – Intellij IDEA – Project setup

W2.6 — Automated Testing of Text UIs

W2.6a — Quality Assurance - Testing - Introduction - What

- Testing: operating a system or component under specified conditions, observing/recording the results, and making an evaluation of some aspect of the system/component



A more elaborate test case can have other details such as those given below.

- A unique identifier : e.g. TC0034-a
- A descriptive name: e.g. vertical scrollbar activation for long web pages
- Objectives: e.g. to check whether the vertical scrollbar is correctly activated when a long web page is loaded to the browser
- Classification information: e.g. priority - medium, category - UI features
- Cleanup, if any: e.g. empty the browser cache.

For each test case you should do the following:

1. Feed the input to the SUT
2. Observe the actual output
3. Compare actual output with the expected output

A test case *failure* is a mismatch between the expected behavior and the actual behavior. A failure indicates a potential *defect* (or a bug), unless the error is in the test case itself.

W2.6b - Quality Assurance - Testing - Regression Testing - What

When you modify a system, the modification may result in some unintended and undesirable effects on the system. Such an effect is called a *regression*.

Regression testing **is the re-testing of the software to detect regressions.** Note that to detect regressions, you need to retest all related components, even if they had been tested before.

Regression testing is more effective when it is done frequently, after each small change. However, doing so can be prohibitively expensive if testing is done manually. Hence, **regression testing is more practical when it is automated**.

W2.6c - Quality Assurance - Testing - Test Automation - What

- Goal of automation: reduce number of test cases to be run manually, not eliminate manual testing altogether
- Importance:
  1. Manual testing of all workflows, all fields, all negative scenarios is time and cost consuming
  2. Difficult to test for multilingual sites manually
  3. Does not require human intervention → can run automated test unattended (overnight)
  4. Increases speed of test execution and test coverage
  5. Manual testing can be boring → error prone
- Criteria for automation:
  1. Executed repeatedly
  2. Very tedious / difficult to perform manually
  3. Time consuming
  4. High risk - business critical

- Unsuitable for automation :

① Newly designed and not executed manually at least once

② Requirements changing frequently

③ Executed on ad_hoc basis

W2.6c - Quality Assurance - Testing - Test Automation
- Automated testing of CLI applications

- Command Line Interface (CLI)
- Use input/output re-direction

> - First, you feed the app with a sequence of test inputs that is stored in a file while redirecting the output to another file.
> - Next, you compare the actual output file with another file containing the expected output.

- < and > operator

```
java MyProgram < input.txt > output.txt
```

- Comparing files :

Windows : `FC output.txt expected.txt`

Unix : `diff output.txt expected.txt`