Unit 7: SOFTWARE DESIGN (& DEVELOPMENT)

Software Design is the process of creating a detailed schematic of a piece of software, that meets the requirements of given a brief. This could be a for building upon a piece of existing software, or

Choosing a Language

If a new piece of software is being designed, there are many different programming languages that you could decide to build it with, each offering different advantages. A good rule of thumb is that the lower the level of language used (The less abstract languages), the more efficient your software can be. For examples you could probably write a relatively fast program in C. But using a higher level language such as C++, Pascal, or Java might mean that the software will be finished faster, as they tend to be faster to write in. Interpreted scripting languages such as Python may also be an option, as the development and support work can happen quickly. Speed and fluidity is not always the number one concern.

Psuedocode

Psuedocode is a notation that describes the step-by-step logic behind a particular algorithm, that is used in the planning stage of software development. There is no universal format, so it is subjective and informal. The primary aim of it is to be readable and understandable to other developers who might read it.

It is often used as an intermediate between the planning stage and writing functional code. Here is an example of a function that returns the factorial of given value, in psuedocode.

```
FUNCTION get_factorial(value)
    factorial = 1
    FOR i in all integers from 1 to i >= value
        factorial = factorial * i

    WRITE "Factorial of "+ value +" = " + factorial
    RETURN factorial
```

Testing

Testing is an important method used for minimizing the risk of damage caused by a new release of a piece of software. Unit tests are scripts specific to a piece

of software that are often run upon a new build. Given known input and output pairs of functions in the software, these tests will ensure that functions in this build give correct outputs, and when given invalid inputs, they error correctly.

Integration tests are scripts that ensure that different components or modules work correctly together. These are important because of two reasons: Because unit tests often can't cover every possibility, and because unit tests only test within in either a single module or selection of modules. Though other modules from outside that scope might depend on functions from within it, so if any behaviour were to change, then these modules should be tested too.

Version Control

The development of software is never a linear process. Often initial decisions on how a tool should work, or what a UI should look like are superceded either feedback from testers or clients, or by unforseen technical limitations or capabilities. So code is constantly being modified, removed and overwritten. This is one reason why tracking development work is critical. It is very cheap to backup old versions of scripts and libraries as are just text files, yet the cost of the time of the developer can be very expensive.

There are a wide variety of solutions available to manage development and version but I will cover two that are commonly used.

Package-Management Systems

Package management systems are often used for building and deploying tools and scripts. These are often centralized systems which stores information about builds of software. Examples of what this information might include are the creator of the build, the time it was built, the compiled build files and information about any dependancies of the software.

Git

Git is a distributed version control system (DVCS) that is used to manage source code development. "Distributed" means that there is no centralized server required. A Git repository is created for each project, which is a Git-managed directory in which source code for that project is stored. I will explain all of the main concepts that are essential for working with Git, and then go through the git commands you will most likely be using if you were using a Git workflow.

Commits

Each developer "commits" his development work to Git and Git saves snapshot changes of source code. Over time, this effectively builds long chains of commits that describe modifications, additions and removals of files, an entire history of the source code, from the creation the repository to its current state.

Tags

Tags are pointers to specific commits, which are often used to denote which commit a version was built from. These can be useful when debugging a particular build of the software. Tags are immutable, meaning that they cannot be overwritten, though they can be explicitly removed and a new tag re-added.

Branches

Git also features branching, a way to divide streams of development by the feature being worked on. These branches can then be merged into the source branch after these have been tested and reviewed. The most common way this is utilized is by having a primary branch, normally named *master* or *release*, which contains reviewed and tested code. Most in-use and/or released software will be built have been from this branch. But before feature branches are merged into this primary branch, there is normally an intermediate step, in which some reviewing and testing may take place. A secondary branch, which might be called *develop* or *testing*, is normally used as this step.

Branches are technically pointers to commits, in the same way as tags. What makes branches different is a branch's commit will update as you add/remove commits, while a tag will remain on a specific commit.

HEAD

HEAD is a special pointer, which points to the commit you are currently working from. All of the files in your repository are a result of all parent commits up to HEAD, therefore HEAD can be thought of as the parent of your next commit. In most cases, it will point the branch you are working on, though there are some cases in which HEAD might not point to a branch's commit. This is called a detached HEAD.

Upstream & Downstream Repositories

A Git repository can be developed upon remotely, by multiple people, with no need for any centralized servers. The first step to working on a remote repository is by using a process called cloning. Cloning a repository copies the commits from the remote Git directory and configures the remote directory as an upstream repository. By default this upstream will be named `origin`. The resultant, local repository is then known as a downstream. New commits and tags on the remote can be pulled on demand, and local changes can be pushed to the remote.

While I mentioned above that Git was a Distributed system, and that there is no need for a centralized server, many workflows do use a centralized server for holding repositories, which developers pull from and push to.

Working with Git

All of the text in these documents that you're reading are stored in Git-controlled markdown files. I show the processes of how contributing to Git projects works using this repository as an example.

Cloning

The first step is to clone a remote repository that you wish to work on. This can be done using the `git clone` command and a path or URL to the remote repository.

```
>>> git clone https://github.com/HBalable/atd-apprentice-module-summaries.git
Cloning into 'atd-apprentice-module-summaries'...
remote: Counting objects: 248, done.
remote: Compressing objects: 100% (93/93), done.
remote: Total 248 (delta 53), reused 12 (delta 3), pack-reused 150
Receiving objects: 100% (248/248), 3.56 MiB | 238.00 KiB/s, done.
Resolving deltas: 100% (123/123), done.
>>> cd ./atd-apprentice-module-summaries
```

`git status` is a very useful command which shows information about your working tree, which is the current state of the source code in your local directory. We should see nothing much here, as we have not yet edited any of the files.

```
>>> git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

We will want to create and switch to a new branch for our changes. We can create a new branch from the current branch (`master`) with the `git branch` command and we'll use the `git checkout` command to switch to it. Let's make a new branch called `example` and check it out.

```
>>> git branch example
>>> git checkout example
Switched to branch 'example'
```

Now let's add a new file to the repository and see what happens.

```
>>> touch example_file
>>> git status
On branch example
```

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        example_file

nothing added to commit but untracked files present (use "git add" to track)
```

You can see that is prompting us to add this file using `git add` to stage it for commit. This extra step is useful for when you might want to commit different changes to different files separately. You only stage the files you want commited. Let's stage our file now.

```
>>> git add example_file
>>> git status
On branch example
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   example_file
```

This file is now staged and ready to be committed. We now just need to use `git commit` command to commit it. We'll use the `-m` arguement to pass a commit message too. These messages should be short sentences that summarise the changes in the commit.

```
>>> git commit -m "added example_file"
[example 97587c8] added example_file
 1 file changed, 0 insertions(+), 0 dele
```

Now we've successfully committed our changes, we can merge them back into the source branch, using the `git merge` command. When working collaboratively on a project, you should never merge to master without at least some testing or reviewing of your code, but I will do it now, just as an example. To do this, we will first have to switch to the `master` branch.

```
>>> git checkout master
>>> git merge example
Updating 6a6fc51..97587c8
Fast-forward
 example_file | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 example_file
```

Now, we just need to push our branches back to the upstream repository, `origin`. We can do that using the `git push` command.

```
>>> git push origin example
Total 0 (delta 0), reused 0 (delta 0)


>>> git push origin master
```

```
Counting objects: 2, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 255 bytes | 255.00 KiB/s, done.
Total 2 (delta 0), reused 0 (delta 0)
To https://github.com/HBalable/atd-apprentice-module-summaries.git
   6a6fc51..97587c8  master -> master
```

And that is how you successfully use Git for development, in it's most simplest form. It is worth mentioning that `git init` is the command used for creating a brand new git repository from the current directory, if you are not working from a remote. There are many other commands that prove to be useful, and plenty of information about them is accessible from the `git --help` command.