Software Engineering

by Ultan

Introduction

Notes from the course, Software Engineering I took. These are notes that are relevant for a general software engineering position.

Other courses, from my degree, that are relevant for a general, entry level software engineering position are Introduction to Programming, Algorithms and Data Structures, Information Management II and Advanced Telecommunications. Notes for these courses are also included in `../`.

These summary notes are based mainly on content from the course, Software Engineering, but also from the course The Missing Semester of Your CS Education at MIT, linked here (license).

The Shell

Type commands into the shell prompt using a terminal. If the shell is asked to run a command that doesn't match one of its programming keywords it consults an environment variable called `$PATH` that lists which directories the shell should search for programs (programs are stored according to the Filesystem Hierarchy Standard). Once found, the program is run.

Using

- Current directory: `.`
- Parent directory: `..`
- Home directory: `~`
- Root: `/`
- Common programs: `pwd`, `cd`, `mv`, `cp`, `mkdir`, `chmod`, `rm`, `rmdir`, `ls`, `man`, `which`, `echo`, `cat`, `sudo`, `curl`
- Curl can be used to use APIs (programmatically access data). Results often returned as JSON
- Common programs with flags: `cd -`, `rm -r`, `ls -l`, `ls -a`
- Modes for a directory:
  - `d` for directory
  - Followed by three groups of three characters for owner, owning group and everyone else respectively
    - `r` for read (`ls` - list contents)
    - `w` for write (`mkdir` - add/remove files to it)
    - `x` to execute (`cd` - enter a directory)
    - `-` indicates that the given principle does not have the given permission
- User ID of the root/super: `0`

Connecting Programs

- Override default streams (input to terminal, output to terminal) using < and >

- `>>` to append to a file
- `|` is a pipe and lets you chain programs such that the output of one is the input of another. This works because most programs accept input from `STDIN` and piping connects `STDOUT` to `STDIN`. If a program does not accept arguments from `STDIN` then use `xargs` e.g. `ls | xargs rm` (delete files in the current directory)

Shell Tools & Scripting

Variables and Strings

```
foo=bar
echo "$foo"
# prints bar
echo '$foo'
# prints $foo
```

Referring to Arguments

- `$0` - Name of the script
- `$1` to `$9` - Arguments to the script
- `$@` - All arguments
- `$#` - Number of arguments
- `$?` - Return code of the previous command
- `$$` - Process ID for the current script
- `!!` - Entire last command including arguments
- `$_` - Last argument from the last command

Commands

- Commands often return output using `STDOUT`, errors through `STDERR` and a return code/exit status (0 good, anything else is an error)
- Exit codes can be used to conditionally execute commands using `&&` and `||`
- Commands can also be separated on the same line using `;`

```
false || echo "Oops, fail"
# Oops, fail

true || echo "Will not be printed"
#

true && echo "Things went well"
# Things went well

false && echo "Will not be printed"
#

false ; echo "This will always run"
# This will always run
```

- Command substitution: `$(CMD)`, getting the output of a command as a variable, e.g. `for file in $(ls)`
- Process substitution: `<( CMD )`, execute the command, put the output in a temporary file and substitute `<()` with that file's name, e.g. `diff <(ls foo) <(ls bar)` will show differences between files in directories `foo` and `bar`

Control Flow and Functions

```
# mcd.sh

mcd () {
    mkdir -p "$1"
    cd "$1"
}

# Run using source mcd.sh (current bash session) or ./mcd.sh (new instance of bash)
```

```
#!/bin/bash

echo "Starting program at $(date)"

echo "Running program $0 with $# arguments with pid $$"

for file in $@; do
    grep foobar $file > /dev/null 2> /dev/null
    # When pattern is not found, grep has exit status 1
    # We redirect STDOUT and STDERR to a null register since we do not care about them

    if [[ $? -ne 0 ]]; then
        echo "File $file does not have any foobar, adding one"
        echo "# foobar" >> "$file"
    fi
done
```

Shebang

Can be used to tell the shell to execute a script with another interpreter instead of a shell command

```
#!/usr/local/bin/python
import sys
for arg in reversed(sys.argv[1:]):
    print(arg)
```

Shell Globbing

```
convert image.{png,jpg}
# Will expand to
convert image.png image.jpg

cp /path/to/project/{foo,bar,baz}.sh /newpath
# Will expand to
cp /path/to/project/foo.sh /path/to/project/bar.sh /path/to/project/baz.sh /newpath

# Globbing techniques can also be combined
mv *{.py,.sh} folder
# Will move all *.py and *.sh files
```

```
mkdir foo bar
touch {foo,bar}/{a..h}
# This creates files foo/a, foo/b, ... foo/h, bar/a, bar/b, ... bar/h

touch foo/x bar/y
diff <(ls foo) <(ls bar)
# Show differences between files in foo and bar
# i.e. outputs:
# < x
# ---
# > y
```

Finding Files

```
# Find all directories named src
find . -name src -type d
# Find all directories named src (note case insensitive)
find . -iname src -type d
# Find all files modified in the last day
find . -mtime -1
```

Alternative: `fd`

Finding Code

```
# Recursively look into directories and look for text files for the matching string
grep -R foobar .

# Invert the match
grep -v foobar myfile.txt
```

Alternative: `rg`

Finding Shell Commands

- `history` e.g. `history | grep foo`
- Starting a command with a leading space won't add it to your shell history
- `CTRL + R`

Other Tools

- Shellcheck
- TLDR Pages

Editors (Vim)

Overview

- Modal editor that is programmable (VimScript, Python) with a programming language as an interface (keystrokes are commands, and those commands are composable)
- Many tools support Vim emulation

Modes

Keystrokes have different meanings in different operating modes. Normal and insert modes are most common

- Normal: For moving around a file and making edits
- Insert: For inserting text
- Replace: For replacing test
- Visual (plain, line or block): For selecting blocks of text
- Command-line: For running a command

Change modes with `<ESC>` to normal mode. Insert mode with `i`, replace mode with `R`, visual mode with `v`, visual line mode with `V`, visual block mode with `<C-V>` and command-line mode with `:`

Basics

Inserting Text

- From normal mode, press `i` to enter insert mode

Buffers, Tabs and Windows

- Vim maintains a set of open files, called buffers. A Vim session has a number of tabs, each of which has a number of windows (split panes). Each window shows a single buffer. A given buffer may be open in multiple windows
- By default Vim opens with a single tab which has a single window

Command-Line

- Type `:` in normal mode
- `:q` to quit Vim
- `:w` to save
- `:wq` to save and quit
- `:e {name of file}` to open file for editing
- `:ls` to show open buffers
- `:help` to open help, e.g. `:help :w`, `:help w`

Movement

- Use movement command to navigate the buffer. Do this in normal mode
- Movements in Vim are also called nouns, because they refer to chunks of text
- Basic movement: `hjkl` for left, down, up, right
- Words: `w` (next word), `b` (beginning of word), `e` (end of word)
- Lines: `0` (beginning of line), `^` (first non-blank character), `$` (end of line)
- Screen: `H` (top of screen), `M` (middle of screen), `L` (bottom of screen)
- Scroll: `ctrl+u` (up), `ctrl+d` (down)

- File: `gg` (beginning of file), `G` (end of file)
- Line numbers: `:{number}<CR>` `{number}G`
- Misc: `%` (corresponding item)
- Find: `f{character}`, `t{character}`, `F{character}`, `T{character}` (find/to forward/backward{character} on the current line, `,` and `;` for navigating matches
- Search: `/{regex}`, `n`, `N` for navigating matches

Selection

- Visual modes: Visual, visual line, visual block
- Can use movement keys to make selection

Edits

- Editing commands that compose with movement commands
- Editing commands in Vim are also called verbs, because verbs act on nouns
- `i` to enter insert mode
- `o` / `O` to insert line above/below
- `d{motion}` to delete e.g. `dw`
- `c{motion}` like `d{motion}` followed by `i`
- `x` to delete character
- `s` to substitute character. Equal to `xi`
- Visual mode plus manipulation by selecting text, `d` to delete it or `c` to change it
- `u` to undo and `<C-r>` to redo
- `y` to copy/yank (some other commands like `d` also copy)
- `p` to paste
- `~` flips the case of a character

Counts

- Combine nouns and verbs with a count e.g. `3w` to move three words forward

Modifiers

- Use modifiers to change the meaning of a noun
- Modifier `i` for inner and `a` for around
  - `ci(` to change the contents inside the current pair of parentheses
  - `ci[` change the contents inside the current pair of square brackets
  - `da'` delete a single quotes string, including the surrounding single quotes

Customising and Extending Vim

- `~/.vimrc` containing Vimscript

- Create the directory `~/.vim/pack/vendor/start/`, and put plugins in there (e.g. via `git clone`)

Advanced Vim

- Search and replace `%s/foo/bar/g` (replace foo with bar globally in a file)
- `:n` and `:prev` to move between files
- `:sp` to split windows

Regular Expressions

- `.` means "any single character" except newline
- `*` zero or more of the preceding match
- `+` one or more of the preceding match
- `[abc]` any one character of `a`, `b`, and `c`
- `(RX1|RX2)` either something that matches `RX1` or `RX2`
- `^` the start of the line
- `$` the end of the line
- Use Regex101 for testing

Command-Line Environment

Job Control

- `Ctrl-c` sends a `SIGINT` signal to a process
- `Ctrl-\` is a `SIGQUIT`
- `kill -TERM <PID>`

Pausing and Background Processes

- Background processes are child processes of the terminal
- `Ctrl-z` ia a `SIGSTP` (the terminal's version of a `SIGSTOP`)
- Use `fg` or `bg` to continue the paused job in the foreground or the background
- `jobs` lists unfinished jobs associated with the current terminal session
- Refer to a process using `%` followed by the job number (displayed by `jobs`)
- Background an already running process? `Ctrl-z` followed by `bg`
- `&` suffix in a command will run the command in the background giving you the prompt back
- To prevent child processes from dying when you close the terminal, run the program with `nohup` or use `disown` if the process has has already been started
- Daemon? A process that is running background, rather than waiting for a user to launch or interact with them

Aliases

- A short form for another command that the shell will replace automatically for you
- To persist across sessions add it to the shell startup files (e.g. `.zshrc`)

Format:

```
alias alias_name="command_to_alias arg1 arg2"
```

Examples:

```
# Make shorthands for common flags
alias ll="ls -lh"

# Save a lot of typing for common commands
alias gs="git status"
alias gc="git commit"
alias v="vim"

# Save you from mistyping
alias sl=ls

# Overwrite existing commands for better defaults
alias mv="mv -i"           # -i prompts before overwrite
alias mkdir="mkdir -p"     # -p make parent dirs as needed
alias df="df -h"           # -h prints human readable format

# Alias can be composed
alias la="ls -A"
alias lla="la -l"

# To ignore an alias run it prepended with \
\ls
# Or disable an alias altogether with unalias
unalias la

# To get an alias definition just call it with alias
alias ll
# Will print ll='ls -lh'
```

Dotfiles

- Many programs are configured using plaintext files known as dotfiles, e.g. `~/.vimrc`
- For bash, change `.bashrc` or `.bash_profile` e.g. many programs ask you to modify your `PATH` environment variable and this can be done in the shell configuration file
- Dotfiles should be in a separate folder, under version control, and symlinked into place using a script
  - Easy installation
  - Portability
  - Synchronisation
  - Change tracking

Remote Machines

Use Secure Shell (SSH).

Connecting

- `user@test.tcd.ie` (can also specify by `IP`)
- SSH keys in `.ssh/` (can generate using `ssh-keygen`)
- Key-based authentication - `ssh-copy-id -i .ssh/id_rsa.pub user@test.tcd.ie` (use locally available keys to authorise login on remote machines)

Executing Commands

- Example local: `ssh foobar@server ls | grep PATTERN` to grep locally the remote output of `ls`
- Example remote: `ls | ssh foobar@server grep PATTERN` to grep remotely the local output of `ls`

Copying Files

- Method one: `cat localfile | ssh remote_server tee serverfile` (`tee` writes the output of `STDIN` into a file)
- Method two: `scp path/to/local_file remote_host:path/to/remote_file`

Port Forwarding

- `ssh -L 9999:localhost:8888 foobar@remote_server`
- `ssh -L sourcePort:forwardToHost:onPort connectToHost` means connect with ssh to `connectToHost`, and forward all connection attempts to the local `sourcePort` to port `onPort` on the machine called `forwardToHost`, which can be reached from the `connectToHost` machine

Configuration

- Can create aliases or use `~/.ssh/config`

Version Control (Git)

Data Model

Snapshots

- History of a collection of files and folders within some top level directory as a series of snapshots (a snapshot is the top level tree that is being tracked)

```
<root> (tree)
|
+- foo (tree)
   |
   + bar.txt (blob, contents = "hello world")
|
+- baz.txt (blob, contents = "git is great")
```

Modelling History

- A directed acyclic graph of snapshots. Each snapshot in git refers to a set of parents (the snapshots that preceded it)

- o is a commit (snapshot) and <— points to the parent of a commit. Note the history branching after commit three

```
o <-- o <-- o <-- o
            ^
             \
              --- o <-- o
```

- Branches can be merged using a merge commit

```
o <-- o <-- o <-- o <---- o
            ^             /
             \           v
              --- o <-- o
```

- Commits in Git are immutable. Edits to the commit history create entirely new commits

Data Model as Pseudocode

```
// A file is a bunch of bytes
type blob = array<byte>

// A directory contains named files and directories
type tree = map<string, tree | blob>

// A commit has parents, metadata, and the top level tree
type commit = struct {
    parent: array<commit> // Most have one parent, but merge can have multiple
    author: string
    message: string
    snapshot: tree
}
```

Objects and Content-Addressing

- An object is a blob, tree or commit

```
type object = blob | tree | commit
```

- In Git data store, all objects are content-addressed by their SHA-1 hash

```
objects = map<string, object>

def store(object):
    id: sha1(object)
    objects[id] = object

def load(id):
    return objects[id]
```

- Important note: When an object references another object, they don't actually contain them in their on disk representation, but have a reference by their hash to where they are in the object store

```
git cat-file -p id
```

References

- Human-readable names for SHA-1 hashes
- Pointers to commits. Unlike objects, references are mutable (can be updated to point to a new commit)

```
references = map<string,string>

def update_reference(name,id):
    references[name] = id

def read_reference(name):
    return references[name]

def load_references(name_or_id):
    if name_or_id in references:
        return load(references[name_or_id])
    else:
        return load(name_or_id)
```

- `HEAD` is a special reference that tells you where you currently are in the history, so when we take a snapshot we know what it is relative to (allowing the parents field of the commit to be set)

Repositories

- On disk, Git stores objects and references. `git` commands map to some manipulation of the commit DAG by adding objects and adding/updating references

Staging Area

- Concept that is orthogonal to the data model
- Allows specifying which modifications should be included in the next snapshot

Git Command-Line Interface

Basics

- `git help <command>`: Get help for a Git command
- `git init`: Create Git repo, with data stored in `.git/`
- `git status`: See what's going on
- `git add <filename>`: Add file to staging area
- `git add <filename> -u`: Add a deleted file to remove
- `git commit -m "message"`: Writes a new commit. Use the imperative, e.g. "Fix bug"
- `git log`: Show a flattened log of history
- `git log —all —graph —decorate`: Visualises history as a DAG
- `git diff <filename>`: Show differences since the last commit
- `git diff <revision> <filename>`: Show differences in a file between snapshots
- `git checkout <revision>`: Updates HEAD and current branch

Branching and Merging

- `git branch`: Show branches
- `git branch <name>`: Creates a branch
- `git checkout -b <name>`: Creates a branch and switches to it (same as `git branch <name>, git checkout <name>`)
- `git checkout -b <new> <existing>`: Creates a branch based off existing and switches to it
- `git merge <revision>`: Merges into the current branch
- `git rebase`: Rebase set of patches onto a new base

Resolving Merge Conflicts

```
>>>>>>> HEAD
    What you have where HEAD is pointing
    ...
=======
    What you are trying to merge
    ...
>>>>>>> branch
```

- Make edits to `file`
- After fixing the conflict use `git add file`, followed by `git merge —continue`
- If you don't want to resolve use `git merge —abort` to cancel the merge

Remotes

- `git remote`: List remotes
- `git remote add <name> <url>`: Add a remote
- `git push <remote> <local branch>:<remote branch>`: Send objects to remote, and update remote reference (create a branch on the remote that is going to be the same as the local branch, e.g. `git push origin master:master`)
- `git branch --set-upstream-to=<remote>/<remote branch>`: Set up correspondence between local and remote branch (once you do this, you can simply write `git push`)
- `git fetch`: Retrieve objects/references from a remote
- `git pull`: Same as `git fetch; git merge`
- `git clone`: Download repository from remote

Undo

- `git commit —amend`: Edit a commit's content message
- `git reset HEAD —file`: Unstage a file
- `git checkout —file`: Discard changes

Other Commands

- `git clone —shallow`: Clone without entire version history
- `git blame`: Show who last edited which line
- `git stash` / `git stash pop`: Temporarily remove modifications to the working directory
- `git config`

Debugging

- Print statements to standard output
- Logging using severity levels (e.g. INFO, DEBUG, WARN, ERROR)
- Most programs write their own logs somewhere in the system
- When printing is not enough, a debugger should be used

```
logger "Hello Logs"
# On macOS
log show --last 1m | grep Hello
```

Metaprogramming

Build Systems

- Makefile. Left-hand side is the target. Right-hand side are the dependencies. The indented block is a sequence of programs to produce the target from the dependencies
- Commands are `make` and `clean`

```
# Sample makefile for fsize program

CC=gcc

fsize: fsize.o
    $(CC) -o fsize fsize.o

fsize.o: fsize.c fsize.h
    $(CC) -c fsize.c
```

Dependency Management

- Versioning: A version number with every release
- Format for semantic versioning: *major.minor.patch*
  - If a new release does not change the API, increase the patch number
  - If you add to the API in a backwards-compatible way, increase the minor version
  - If you change the API in a non-backwards-compatible way, increase the major version
- Lock file: Lists the exact version you are currently depending on for each dependency
- Vendoring: Copy all your code of your dependencies into your own project

Continuous Integration Systems

- Stuff that runs whenever your code changes
- Add a file to a repository that describes what should happen when various things happen to that repository e.g. when someone pushes code, run the test suite
  - Event triggers
  - Spin up VM(s)
  - Run commands in the recipe

  - Note down the results somewhere

Testing

- Test suite: All tests
- Unit test: A micro test that tests a specific feature in isolation
- Integration test: A macro test that tests a larger part of the system to check that different features or components work together
- Regression test: A test that implements a particular pattern that previously cause a bug to ensure that the bug does not resurface
- Mocking: Replacing a function, module or type with a fake implementation to avoid testing unrelated functionality e.g. mock the network

Markdown

- – or * or + is an unordered list item
- Numbers are used for an ordered list item
- Indent four spaces to add another element within a list while preserving list continuity
- * for italics
- ** for bold
- # for headings
- “ to surround words making them code font
- Indent by four spaces for a code block (with extended Markdown you can use ```)
- `[text](url "tooltip")` for links
- `![alt text](url)` for images
- Use a blank line to create paragraphs
- End a line with two or more spaces and return to create a line break
- > to create a block quote
- *** to create a horizontal rule
- <> enclosing an email address
- \ for escaping characters

Virtual Machines and Containers

- Virtual machine execute an entire OS stack, including the kernel
- Containers avoid running another instance of the kernel and instead share the kernel with the host
- Containers have weaker isolation and only work if the host runs the same kernel

Android Studio

- Lots of features for writing Android applications including
  - Emulator
  - Device file explorer
  - Layout editor
  - Screen recorder
  - Gradle build system