COMP1531

3.3 - Verification & Validation

Verification

Verification in a system life cycle context is a set of activities that compares a product of the system life cycle against the required characteristics for that product. This may include, but is not limited to, specified requirements, design description and the system itself.

Validation

Validation in a system life cycle context is a set of activities ensuring and gaining confidence that a system is able to accomplish its intended use, goals and objectives.

*ISO/IEC/IEEE 29148:2018

Verification

The system has been built right

Validation

The right system has been built

*by intuition

Formal Verification

- Proving (via Mathematics) that a piece of software has certain desirable properties
- Treats the software, or the algorithms implemented in the software, as a mathematical object that can be reasoned about.
- Typically involves tools like proof assistants, model checkers or automatic theorem provers.
- **Not something we cover in this course**

Formal Verification

- Tends to have a high cost in terms of effort
- E.g. to verify a microkernel
  - it took ~20 person years
  - and ~480,000 lines of proof script
  - for ~10,000 of C

What is testing anyway?

“Testing shows the presence, not the absence of bugs” — *Edsger W. Dijkstra*

Unit testing

ISTQB definition:

The testing of individual software components

Method:

White-box

Who:

Software Engineers

Integration Testing

ISTQB definition:

Testing performed to expose defects in the interfaces and in the interactions between integrated components or systems.

Method:

White-box or Black-box

Who:

Software Engineers or independent testers

System Testing

ISTQB definition:

The process of testing an integrated system to verify that it meets specified requirements.

Method:

Black-box

Who:

Normally, independent testers

Acceptance Testing

ISTQB definition:

The process of testing an integrated system to verify that it meets specified requirements.

Method:

Black-box

Who:

User or Customer

How do we know if our tests are good?

Coverage

- Test Coverage: a measure of how much of the feature set is covered with tests
- Code coverage: a measure of how much code is executed during testing

Example: Leap years

```
def is_leap_year(year):
    if year % 4 != 0:
        return False
    elif year % 100 != 0:
        return True
    elif year % 400 != 0:
        return False
    else:
        return True
```

Coverage.py

- Measure code coverage as a percentage of statements (lines) executed
- Can give us a good indication how much of our code is executed by the tests
- ... and most importantly highlight what has **not** been executed.

Example: *spoiler*

Checking code coverage

- Run Coverage.py for your pytests:

```
python3-coverage run --source=. -m pytest
```

- View the coverage report:

```
python3-coverage report
```

- Generate HTML to see a breakdown (puts report in htmlcov/)

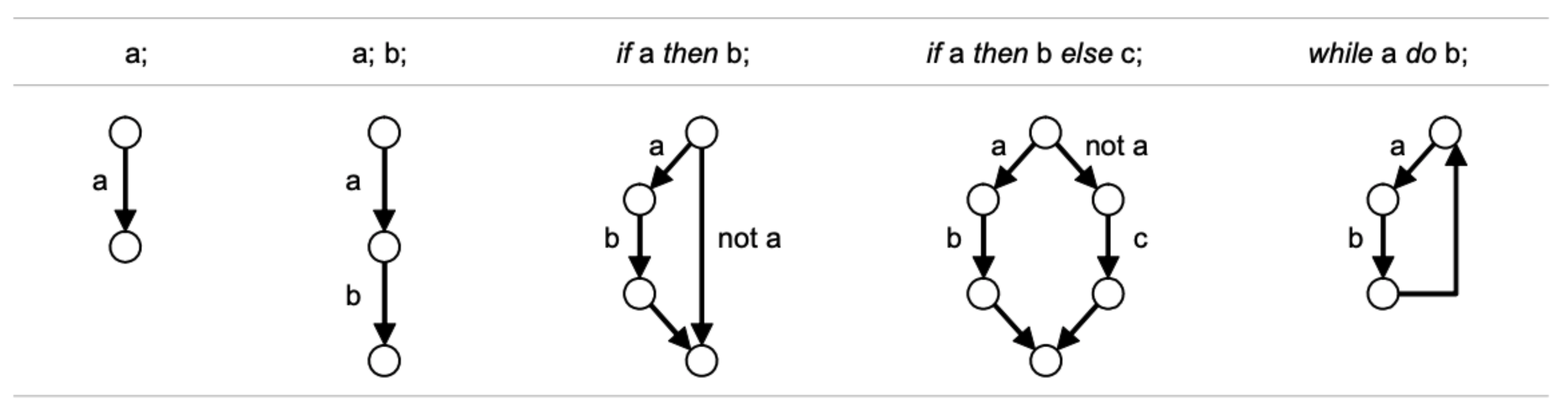
```
python3-coverage html
```

Spoiler

Spoiler

Branch coverage checking

- For lines that can potentially jump to more than one other line (e.g. if statements), check how many of the possible branches were taken during execution
- Can be done with the --branch option in Coverage.py
- Sometimes referred to as edge coverage



Does code coverage imply test coverage?

What is the right level of code coverage?

Summary

- Code coverage is useful
- It's more important to look at what's not covered than the coverage percentage
- Branch coverage is a more accurate measurement so you should use it instead of statement coverage
- Like all measurements, it's important to understand what meaning to attach to it

What is good style?

Programs must be written for people to read, and only incidentally for machines to execute – *Abelson & Sussman, "Structure and Interpretation of Computer Programs"*

Style

- Ultimately about readability and maintainability
- Style guides give rules of thumb and conventions to follow
- ...but good style is ultimately hard, if not impossible, to measure
- That said, tools can be a lot of help

There are a lot of tools in modern software engineering

Pylint

- An external tool for *statically* analysing python code
- Can detect errors, warn of potential errors, check against conventions, and give possible refactorings
- By default, it is **very** strict
- Can be configured to be more lenient

Controlling Messages

- Disable messages via the command line

```
$ pylint3 --disable=<checks> <files_to_check>
```

- Disable messages in code; e.g.

```
if year % 4 != 0: #pylint: disable=no-else-return
```

- Disable messages via a config file
  - If a .pylintrc file is in the current directory it will be used
  - Can generate one with:

```
pylint3 <options> --generate-rcfile > .pylintrc
```