UC
UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

Fabian Gilson

Software Engineering II

SENG301 - Testing software

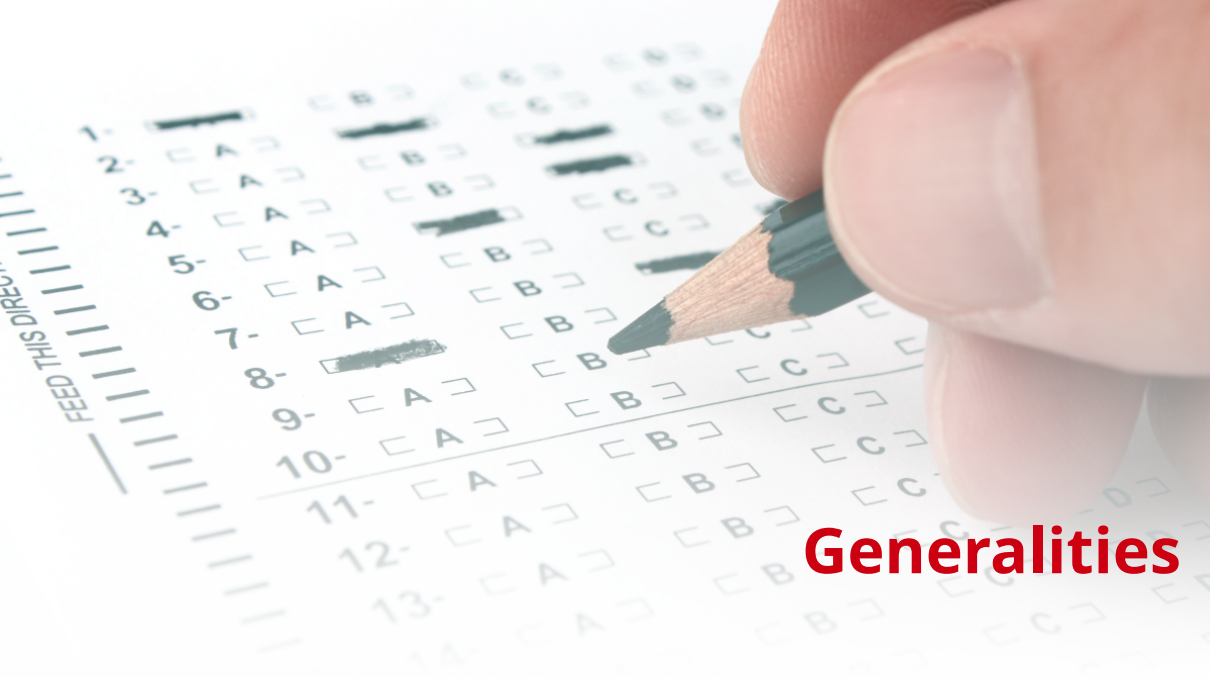
Lecture 7 - Semester 1, 2020

University of Canterbury | Te Whare Wānanga o Waitaha
Computer Science and Software Engineering

As a rule, software systems do not work well until they have been used, and have failed repeatedly, in real applications.

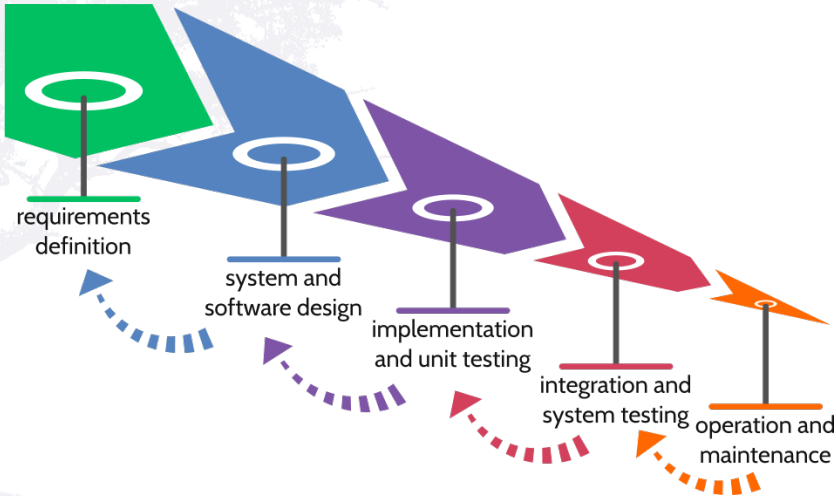
– David Parnas

Communications of the ACM 33, p.636 –1990

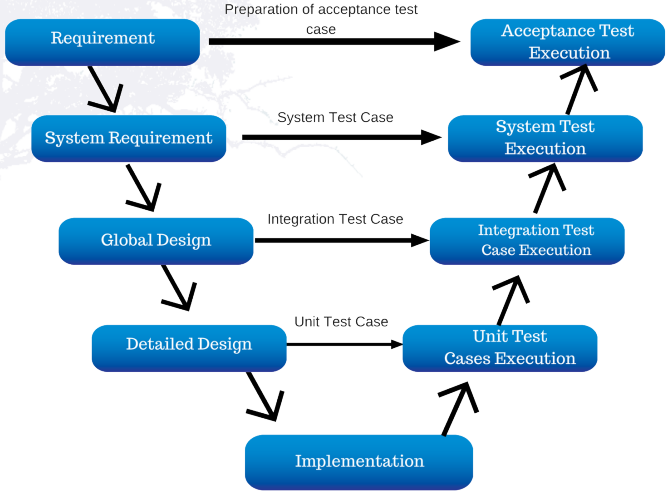


Generalities

Remember the Waterfall (RIP)



From late to ongoing test writing



from *http://www.codekul.com/blog/methodologies-software-testing-life-cycle/*

Objectives of testing

" Testing can only show the presence of errors, not their absences. "

Dijkstra, "The Humble Programmer", 1972

Two main objectives

validation demonstrate the software fulfils its requirements

verification identify erroneous behaviour of the system

Determine *"fit for purpose"* state

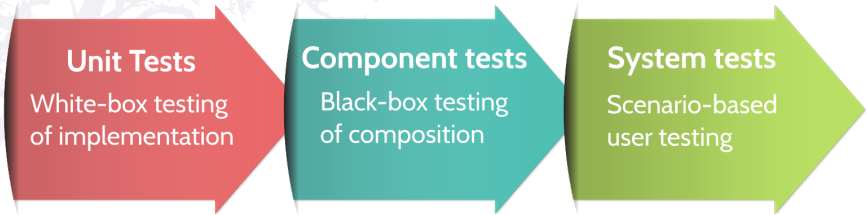
purpose safety-critical systems are tested differently, *e.g.*, formally validated

expectations users may accept more instability for young software

marketing time-to-market and price also influence testing & validation

Testing phases

Staged testing



Unit testing

Any piece of code should be tested

- every feature should be **testable** and **tested**
- use **fake/simulated** inputs to your methods
- **avoid** human or third-parties interactions

On top of explicit test classes, your methods should prevent missuses

- code assertion are primordial for **regression tests**
- explicit verifications of **pre/post conditions** at component boundaries

Remember to adopt a sceptical mind

- identify (mathematical) **domains for values** supposed to have the same effects
- think about **edge cases**, *i.e.* values at domain's boundaries

Component testing

Testing of identifiable and isolated part of a system

→ hidden and **interchangeable** implementations

Various types of interfaces

application behaviour accessible through *operations* (*e.g.,* methods) calls

memory shared block of memory between processes

message passing of data between operations through a communication medium

On top of being sceptical on input data

- make components fail, *aka* check for **differing failures**
- **stress testing**, *i.e.* call or message overflow
- if a **call order** exists, try to call operations in any order

System testing

Finally, the last testing stage

- integrate **third-party** components or systems
- should be performed by *"dedicated"* testers, and surely **not only** by developers

Scenario-based testing

- main usages, *i.e.* full **interaction flows**, should be modelled
- start **from** (graphical) **user interface** interactions and go **to data** layer

Trace and **record** tests executions, *i.e.* a wiki or any other form

- **input** values, **expected** output, **observed** output
- metadata like **who**, **when** and associated **issue tracking id**

Agile way of doing

Agile staged testing

Let's write some tests all together

```java
private static int greatestCommonDivisor(int n1, int n2) {
  n1 = n1 > 0 ? n1 : -n1;
  n2 = n2 > 0 ? n2 : -n2;
  while(n1 != n2) {
    if (n1 > n2) {
      n1 -= n2;
    } else {
      n2 -= n1;
    }
  }
  return n1;
}
private static long factorial(int n) {
  long factorial = 1;
  for (int i = 1; i <= n; i++) {
    factorial *= i;
  }
  return factorial;
}
```

Smarter unit testing

Assertion clauses

- check **incoming** parameters' values
- ensure **invariant** properties remain valid
- `assert` are useful for regression tests

→ remember lab 1 where `assert` clauses have been shown

Guideline-based testing

- identify **common** programming **mistakes**
- ensure tests are performed on these **particular aspects**
- *e.g.*, database interactions, unmanaged `null` values

Think about Test-Driven-Development Beck, *Test Driven Development: By Example*, 2002

→ write the tests then implement your method

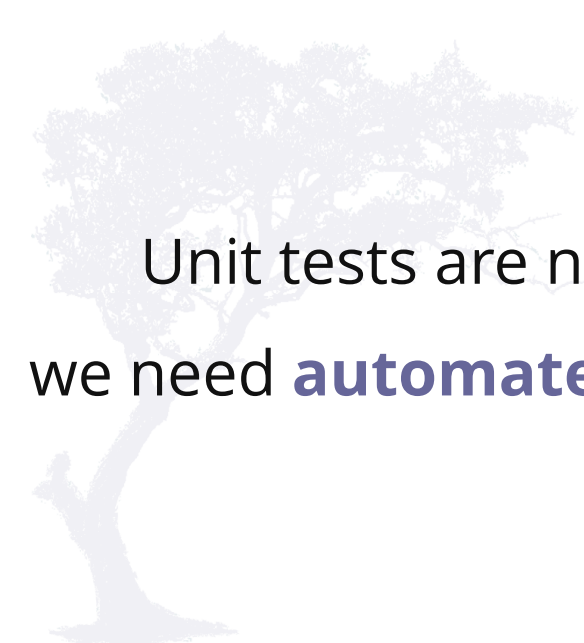
Integration testing at each commit

Each commit to the main branch of the code repository

- triggers a full run of **all unit tests**
- prevents system **regression**

Remember lectures 5 and 6

- avoid third party components at this stage
- if build time is too long, use subset of tests (remember **guideline testing**)
- full building during off-peak time of Continuous Integration system
- built binaries are passed to subsequent stages
- any successful build becomes a **release candidate**



Unit tests are not the sole truth,

we need **automated acceptance tests**

Acceptance criteria as user testing

User stories are always accompanied by acceptance criteria

- do you remember about **INVEST** ?
- run of testing scenarios in a *business-minded* mode

But what do we need to automate them?

- define **application interfaces**, *i.e.* isolate UI
- use **dependency injection**, *aka* *inversion of control*)
- fake minimal implementations for dependencies, *i.e.* **stubs**
- fake interactions with the database, *i.e.* **mocking**
- split **asynchronous** scenarios into synchronous pieces
- keep the tests **simple** and independent
- think about **faking time** with proper separation of concerns

Automated acceptance test

Should use same *path* as users

- using playback tools is not always a possibility, *e.g., Selenium*, *Serenity*
- testing directly on GUI may be **time-consuming** and painful

→ A decoupled UI of any logic increases **testability**

Using *Cucumber* for automated acceptance tests (*cfr.* lab 4)

- annotate methods with *given… when… then…* clauses
- refer to **run reports** in case of problems

Even if playback tool is used, both set of tests should be written

- maintaining **playback scripts** is more difficult and **time consuming**
- well designed API tends to change **less frequently**

Capacity testing

Some non-functional requirements are difficult to test

- **quality attributes** are a major part of the system's requirements
- maintainability and auditability are **non testable** properties **as such**

Semi-automatic testing **capacity**-based requirements

- response time may be expressed as **user stories** or required **system features**
- prioritised and evaluated through acceptance criteria

Capacity testing environment

- **clone** of production system, but **not always feasible**
- **canary deployment** to monitor increases in charge

⚠ benchmark and **scaling factors**

Sprint reviews on top of scenario-based testing

As sprint reviews must be planned and prepared (*cfr. Lecture 3*)

- ensure **all sprint scenarios** are running smoothly
- may need to **rework** some stories

  but avoid trying to fix nasty bugs at the very last minute

  → **rollback** to a previous **release candidate**

Sprint reviews are the last opportunity to ensure everything is ok

- do a *last-minute-verification*, *i.e.* **smoke tests**
- make you more **confident** for the upcoming sprint review

Remember to track testing results appropriately, also for smoke tests

- keep **centralised** any test execution results
- tracking sheet is a **living** document


TO BE
CONTINUED...

Next episode

Let's talk a bit more about system availability and reliability

Once upon a time, the Internet was born

And we will talk about resilience engineering too

... during next lectures