# TDDE45 - Lecture 7: Testability

Adrian Pop and Martin Sjölund

Department of Computer and Information Science
Linköping University

2023-10-03

# Part I

# Testing

# What is testing?

The process that ensures that the code, component, module or application works according to the requirements. Testing properties:

- Systematic
- Black box
- White box

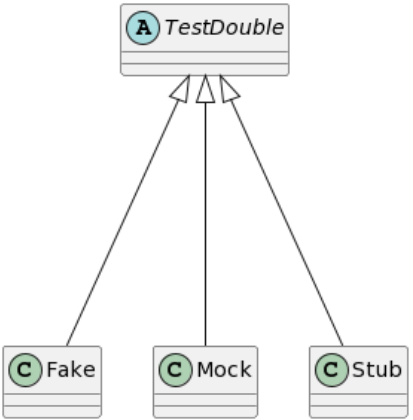
This course is not a course primarily in Software Testing (TDDD04 – HT1).

# Testing Crash Course - Type of Tests

- Unit Tests - specific methods and logic of the code, edge cases
- Feature Tests - tests the functionality of the component (does it meets the requirements?)
- Integration Tests - tests the entire application end to end
- Performance Tests - tests the efficiency of a piece of code (method or the entire application)

# Testing Crash Course - Vocabulary

- Test doubles - used instead of other objects
- Fakes - objects that have working implementations.
- Mocks - are objects that have predefined behavior.
- Stubs - are objects that return predefined values.



# Testing Crash Course - Fakes

Fakes

- objects that have working implementations.
- implement the same interface as a real object but takes shortcuts to improve performance.
- usually used when we need to test something that depends on an external service or API and we don't want to make actual calls to that service.

Example: in memory database.

# Testing Crash Course - Mocks

Mocks

- objects that have predefined behavior.
- they register calls they receive, allowing asserts on how we use them in the code.
- they do not have working implementations but instead they have pre-programmed expectations about how they will be used in the code.
- usually used to test the behaviour of our code rather than its input (calling the dependencies in the expected way).

Example: object that returns a specific value when called with certain arguments.

# Testing Crash Course - Stubs

Stubs

- objects that return predefined values.
- they do not have working implementations.
- not programmed to expect specific calls but they return values when called.
- used to provide the data (hard-coded or dynamically generated) needed for the code to run.

Example: object returning some predefined data expected from a service.

# Testing Crash Course - Choosing the Double

How to choose the test double depending what we are testing. Choose the simplest that gets the job done.

- Fakes - use when you want to improve performance and lower the resource consumption.
- Mocks - use when you want to verify the behaviour of the code.
- Stubs - use when you need to provide data for the code or the test to run.

# When we design for testability, we use white-box testing

- We write the source code
- We make sure that the code can be tested

# Injecting bad behavior into a model

- Testing needs both ☺ and ☹ paths
- How do we test these paths?
- How do we know we have tested all paths?

# Automatability

- To which degree can you automate the test?
- How long time does it take to create an automated test?
- How long time does it take to manually test it?

Graphical user interfaces usually have a high cost of automated testing.

# Controllability

- Can we control the tested object?
- We might need access to other objects that need to change

# Isolateability

- How many other objects are needed to test the object?
- Ideally zero dependencies.

# Understandability

- Is the object documented?
- Is the code self-explaining?
- Will the tester be able to find all edge cases?

# Homogeneity

- Are all the modules written in the same language?
- Are you using the same framework everywhere?
- The more differences, the more different techniques and testing frameworks you might need

# Observability

- How do we know that the test did what it should?
- Private member variables? Reflections API? Member functions that you can infer a value from?
- Visible side effects? File was created? Test that. Read its contents.
- Hidden side effects? ☹
- Extend the class with functions keeping track of what you need to test.

# Separation of concerns

- More functionality in a class means more things to test.
- Encapsulated classes.
  - Modularize the software.
  - Can hide implementation details from other functionality.
- A smaller interface means fewer functions you need to test.
- See also: Single Responsibility Principle.

# Dependency Injection / Inversion of control

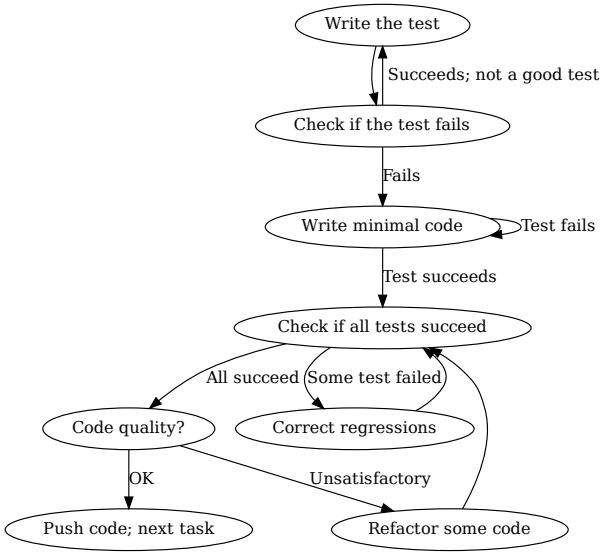
- Allows a way to pass objects (dependencies) to another object
- Helps automatability (makes it easier to construct the tested object's dependencies)
- Allows fakes or mocks to be used
- You can inject dependencies that improve observability of the internal state of the object – you can now access its internal dependency by keeping a pointer to it

```
public class Injector {
  public static void main(String[] args) {
    // Build the dependencies first
    Service service = new ExampleService();
    // Inject the service, constructor style
    Client client = new Client(service);
    // Use the objects
    System.out.println(client.greet());
  }
}
```

# Test-driven development (TDD)

- Requires automation
- Requires short development cycles
- Focused on small unit tests – not larger functional tests
- Test code should be larger than the code under test (need to setup fakes, mocks, etc)
- Still need a testing team to get a different set of eyes
- Many tests; expensive to maintain
- Results in debuggers being less needed

# Test-driven development (TDD) Cycle



# Behavior-driven development (BDD)

Similar to TDD, but focuses on behavior instead of unit tests. Typically, a DSL is used to describe the test:

```
Specification: Stack

When a new stack is created
Then it is empty

When an element is added to the stack
Then that element is at the top of the stack

When a stack has N elements
And element E is on top of the stack
Then a pop operation returns E
And the new size of the stack is N-1
```

# Design for testing (analogy with IC design)

How do you know that hardware is correct?

- It's black box testing (input/output)
- The underlying source code is available, but not very useful for testing
- Add additional testability features to the integrated circuit
- Test the circuit during manufacturing
- Possibly also supports troubleshooting by consumer via JTAG connector

# What you can do today

Use existing development services such as github (or others):
[https://docs.github.com/en/actions/automating-builds-and-tests

- Continuous integration - build and test on various OS and hardware
- Use Test Automation Frameworks and Mock Testing Frameworks
  - Selenium, Cypress, Playwright, WebDriverIO, TestCafe, NightwatchJS, Appium (Mobile), Cucumber (BDD)
  - Mockito, EasyMock, WireMock, MockWebServer, JMockit, PowerMock,
- Build for various languages:
  - Go, Java (with Ant, Gradle, Maven), .NET
  - Node.js (JavaScript/TypeScript)
  - Python, Ruby, Swift, PowerShell
  - Julia - testing and code coverage
  - Xamarin (.NET mobile)

# Testing OpenModelica

`https://openmodelica.org`

- For each pull request (PR): `https://github.com/OpenModelica/OpenModelica`
  - we build it using gcc & clang on Linux (autoconf & cmake)
  - we can also build it on Windows [32 and 64 bit] & Mac [x86_64 and M1] when testing large PRs
  - we run 4000+ tests
  - we partially test the GUI using Qt testing
  - we export models and test that they work with external tools
- Each night we run coverage of 78 Modelica libraries - 16332 models
  - `https://libraries.openmodelica.org/branches/overview.html`
- Each night we build nightly-builds
  - various Linux distros (arm, x86_64)
  - Windows (32 and 64 bit)
  - Mac OS (x86_64, M1)

www.liu.se