Menti Q&A QR

Testing

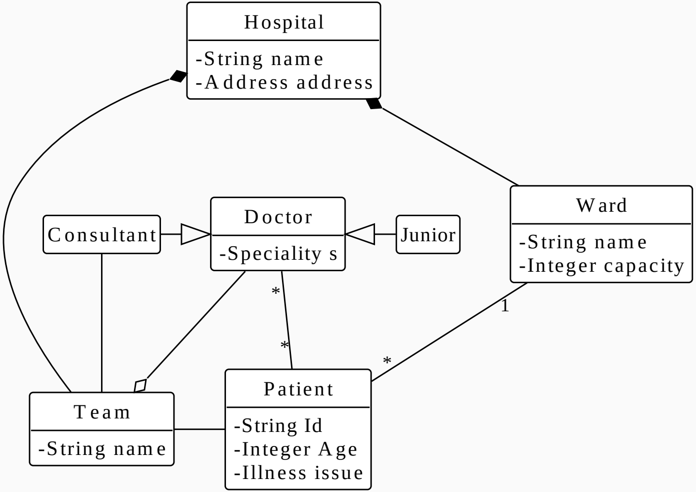
Blair Archibald

Recap

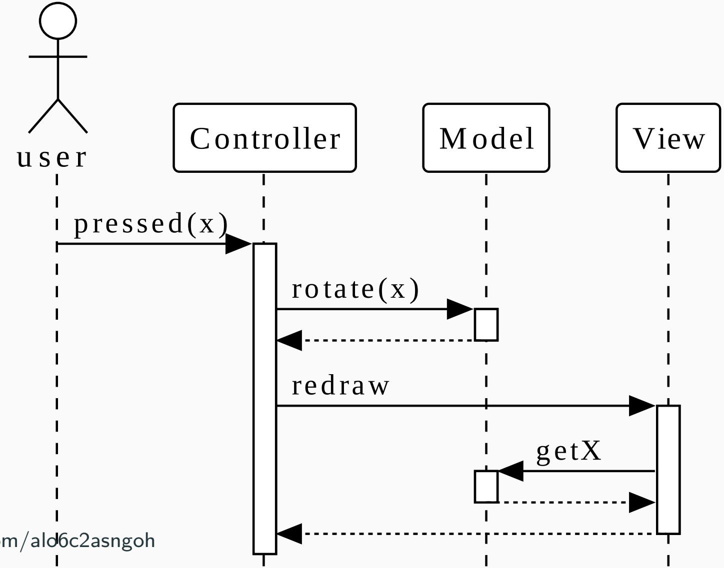
We've explored:

- How to reason about designs using coupling and cohesion
- OOP constructs to help with design: interfaces, visibility etc
- UML:
  - Class Diagrams
  - Sequence Diagrams

Recap: UML Class Diagrams



Recap: UML Sequence Diagrams



Today: We will explore

- How do we know our code is correct?
  - What does correct even mean?
- Different Types of testing
- Different Levels of Rigour
- How to write good test cases
- Test-Driven Development

How do we know it works?

- Even the best designed code is useless if it doesn't work
  - But how do we know it works?

How do we know it works?

- Even the best designed code is useless if it doesn't work
  - But how do we know it works?
- By "works" we mean "meets a specification"
  - Requirements capture gives the specification

What is Testing

Specification (User/System Requirements) $\xleftarrow{?}$ Implementation (Code)

Testing

Evidence for this link

Levels of Testing

- A program might be built of smaller components
  - A specification can also be built of smaller specifications
- For example: a class has a specification, i.e. what it must do
- Is my `List` implementation working as expected?
  - Items must come out in order etc

Different Types of Testing

- Each component/class does what it should: **Unit Tests**
- Components work together correctly **Integration Tests**
- The acceptance criteria (semester 1) are met **Acceptance Tests**

Different Types of Testing

- Each component/class does what it should: **Unit Tests**
- Components work together correctly **Integration Tests**
- The acceptance criteria (semester 1) are met **Acceptance Tests**
- Often:
  - *Developers* write unit tests
  - Integration/Acceptance is a specialist Testing/Quality Assurance team

Different Types of Testing

- Each component/class does what it should: **Unit Tests**
- Components work together correctly **Integration Tests**
- The acceptance criteria (semester 1) are met **Acceptance Tests**

- Often:
  - *Developers* write unit tests
  - Integration/Acceptance is a specialist Testing/Quality Assurance team

We focus on **unit tests** in this course

Different Levels of Rigour

- Different levels of rigour to define "works":
  - Worked when I stepped through by hand
  - Automated, but hand written, tests for some specific inputs/states
  - Randomised automated tests for some properties: *Property based testing*[1]
  - Full mathematical verification: A *proof* the code meets spec for all inputs

[1] Not discussed here, but look into "Quickcheck" if interested

Different Levels of Rigour

- Different levels of rigour to define "works":
  - Worked when I stepped through by hand
  - Automated, but hand written, tests for some specific inputs/states
  - Randomised automated tests for some properties: *Property based testing*[1]
  - Full mathematical verification: A *proof* the code meets spec for all inputs
- Automated, hand written, tests most common (currently! **We will focus here**)
- Property tests getting more common and a hot research topic
- Verification often seen as "too difficult"; Currently useful for safety critical programs, but I predict you will see more of this in future for other domains

[1] Not discussed here, but look into "Quickcheck" if interested

Test Cases

Tests have three main components

- **Givens**: What state is the system in before testing
- **Operations**: What do I do to the system
- **Assertions**: What state do I expect the system is now in

Test Cases

Tests have three main components

- **Givens**: What state is the system in before testing
- **Operations**: What do I do to the system
- **Assertions**: What state do I expect the system is now in

- **Given** an array a of n integers
- **Operation(s)** `a.sort()`
- **Assertions**: a still has n integers, **and** each $a[i] <= a[i+1]$

Unit Testing

- Unit testing are tests for a single function/class/component
- Accepted wisdom is that every class has:
  - Associated test class
  - Test cases for each method
- These tests are **fast**
  - Run them after every change/before commits
  - Sorting 5m element arrays is probably a bad test case!
- Usually cannot commit code without tests
  - It will fail a code review instantly (maybe even automatically)

Test Data

Specifying the givens can be tricky

- Given **an** array a of n integers
  - But which specific array?

- We actually want a set of test cases for a range of givens
  - Carefully chosen to test as much of the system as possible
  - Differences in levels of testing occur here:
    - Verification = "forall" possible inputs
    - Property = for some large set of inputs
    - Hand written = for these specific inputs

Test Data

What makes good test data? Data that matches

- Common cases: if the system expects 200 elem arrays then test that
- Extreme/Edge cases:
  - Empty array
  - Single element array
  - Non-integer array
  - Ready sorted array[2]

[2] Of course you need to sort this with a different function to the one you are testing!

Exercise: Testing A Date Class

```java
public class Date {
  private int day, month, year;
  public Date(int d, int m, int y) { ... }

  // We want to test this function
  public Date addDays(int d) { ... }
}
```

Take 3–5 Minutes and come up with some possible test cases for `addDays`

Remember the goal is to stress-test the function

Exercise: Testing A Date Class

Case	Given	Op	Expect/Assert
Small addition	1/1/2023	+5	6/1/2023
Over month boundary	29/1/2023	+5	3/2/2023
Over year boundary	29/12/2023	+5	3/1/2024
Feb special cases	28/2/2024	+5	5/3/2023
Idempotency	1/1/2023	+0	1/1/2023
Negative	5/1/2023	-1	?

Exercise: Testing A Date Class

Case	Given	Op	Expect/Assert
Small addition	1/1/2023	+5	6/1/2023
Over month boundary	29/1/2023	+5	3/2/2023
Over year boundary	29/12/2023	+5	3/1/2024
Feb special cases	28/2/2024	+5	5/3/2023
Idempotency	1/1/2023	+0	1/1/2023
Negative	5/1/2023	-1	?

- Negative needs more input from the spec:
  - Throws an exception (`InvalidArg`)
  - Does nothing: 5/1/2023 + (-1) = 5/1/2023
  - Allows going back in time: 5/1/2023 + (-1) = 4/1/2023

Testing might uncover questions *about the spec*

Interacting with the Environment

- Classes don't operate in isolation
  - The whole point of OOP is that *interaction of objects solves the problem*!
- To test a class we need the dependencies in the correct states
  - The *given* is the full state that affects this case

```
void testPayment(int amount) {
  VisaCreditCard c = new VisaCreditCard();
  Account acc = new Account(c);
  assert(acc.getCost() == 0);
  c.doPayment(amount);
  assert(acc.getCost() == amount);
}
```

Interacting with the Environment

```
void testPayment(int amount) {
  VisaCreditCard c = new VisaCreditCard();
  Account acc = new Account(c);
  assert(acc.getCost() == 0);
  acc.doPayment(amount);
  assert(acc.getCost() == amount);
}
```

- What if `CreditCard` integrates with Visa directly?
  - We might have just charged a real credit card!
- Often we need *mock* dependencies
  - That have the right interfaces
  - But do “fake” behaviour

Mocks

Mocks are another reason programming to **interfaces** is so important!

```
void Account(VisaCreditCard c) { ... }
```

Hard to change!

Mocks

Mocks are another reason programming to **interfaces** is so important!

```
public interface CreditCardSupplier {
  public void charge(int amount);
}

public VisaCreditCard extends CreditCardSupplier { ... }
public TestCreditCard extends CreditCardSupplier { ... }

void Account(CreditCardSupplier c) { ... }
```

Can now test safely

Mocks in Practice

- Mocks are so common there are *libraries* that can create them dynamically
  - This example is from “Mockito”

```
VisaCreditCard fakeCard = mock(VisaCreditCard.class);
when(fakeCard.charge(100)).thenReturn(true);

assert(mockedList.charge(100) == true);
```

Note: You are expected to know how to do this manually for this course, not how to use Mockito or similar

Test Coverage

- **Test Coverage**: a metric to (try to) determine "how well tested code is"
  - Out of all lines-of-code, what percentage does the unit tests cover

Test Coverage

- **Test Coverage**: a metric to (try to) determine "how well tested code is"
  - Out of all lines-of-code, what percentage does the unit tests cover

```
int doSomething(int x) {
  if (x == 42) {
    println("Hidden feature");
    return 0;
  }
  println("Normal Path");
  return 1;
}
```

```
void testDoSomething() {
  assert(doSomething(1) == 1);
}
```

Covers 3 of 5 statements (60% coverage)

```
void testDoSomething() {
  assert(doSomething(1) == 1);
  assert(doSomething(42) == 0);
}
```

Covers 5 of 5 statements (100% coverage)

Test Coverage

- **Test Coverage**: a metric to (try to) determine "how well tested code is"
  - Out of all lines-of-code, what percentage does the unit tests cover

```
int doSomething(int x) {
  if (x == 42) {
    println("Hidden feature");
    return 0;
  }
  println("Normal Path");
  return 1;
}
```

```
void testDoSomething() {
  assert(doSomething(1) == 1);
}
```

Covers 3 of 5 statements (60% coverage)

```
void testDoSomething() {
  assert(doSomething(1) == 1);
  assert(doSomething(42) == 0);
}
```

Covers 5 of 5 statements (100% coverage)

Caveat: 100% coverage does not mean **no errors** or **definitely meets specification**

Tests only show the Presence of Bugs

Worth thinking about this:

> "program testing can be used very effectively to show the presence of bugs but never to show their absence."[3] E. W. Dijkstra[4]

- Important: Dijkstra does not include verification as a form of testing (like I did)
  - Verification *does* let you show absence
- For now, "Testing" is still heavily used despite this shortcoming

[3] From https://www.cs.utexas.edu/users/EWD/transcriptions/EWD03xx/EWD303.html
[4] One of the most famous Computing Scientists: worth looking at some of his writings/talks!

Testing Recap

- Testing provides *evidence* that an implementation meets a specification

- Types: unit, integration, acceptance
- Levels: hand, automated, property based, verification

- Test Cases: Givens—Operations—Assertions

Test Driven Development (TDD)

- Testing is so fundamental it is the core of some methodologies
  - Particularly in Agile: Spec Changes $\implies$ Test Changes
  - Means you need well designed tests
- TDD Loop:
  - Write a *failing* test (means you have to define the calling interface to use)
  - Write the *simplest* code that makes the test pass
  - Refactor the code to improve design
  - Also known as “red-green-refactor”

TDD Example: Password Verifier

Lets write a class that verifies passwords meet some criteria

- Start by defining a test

```
public void testEmptyPasswordIsNotStrong() {
  String pass = "";
  PasswordVerifier v = new PasswordVerifier();
  assert(!v.isStrong(pass));
}
```

TDD Example: Password Verifier

Lets write a class that verifies passwords meet some criteria

- Start by defining a test

```
public void testEmptyPasswordIsNotStrong() {
  String pass = "";
  PasswordVerifier v = new PasswordVerifier();
  assert(!v.isStrong(pass));
}
```

- Fails to compile since `PasswordVerifiver` doesn't exist!

TDD Example: Define Minimal Working Example

```
public class PasswordVerifier {
  public isStrong(String pass) {
     return false;
  }
}
```

- Test now compiles
  - It also is successful (green)
- No refactoring needed since it's so simple

TDD Example: Another test

We then write another test:

```java
public void testPasswordLessThan8CharactersIsWeak() {
  String pass = "123456";
  PasswordVerifier v = new PasswordVerifier();
  assert(!v.isStrong(pass));
}
```

- This one still passes
    - So isn't really the best next TDD case!
    - But still useful to have

TDD Example: Another test

```java
public void testPasswordMoreThan8CharactersIStrong() {
  String pass = "12345689";
  PasswordVerifier v = new PasswordVerifier();
  assert(v.isStrong(pass));
}
```

- Fails! So we need to fix the code so it passes

TDD Example: Fixing the Code

Tests are passing, so we write another

```
public class PasswordVerifier {
  public isStrong(String pass) {
     if (pass.length() >= 8) { return true; }
     return false;
  }
}
```

- Passes
  - Not much to refactor
  - We might promote the magic number 8 to a static field

```
private final static minLen = 8;
```

TDD

- Continue with the red-green-refactor
  - Stop when you are happy there's enough tests to show specification is (likely) met

TDD Caveats

- Some people really like TDD
  - Makes code that is "easy to test"
  - Not clear "easy to test" = "best design"
    - But it's one way to think about design
- Sometimes feels quite extreme
  - Could have jumped to the implementation of the password verifier quicker
  - Most people seem to do "something like TDD" but not religiously

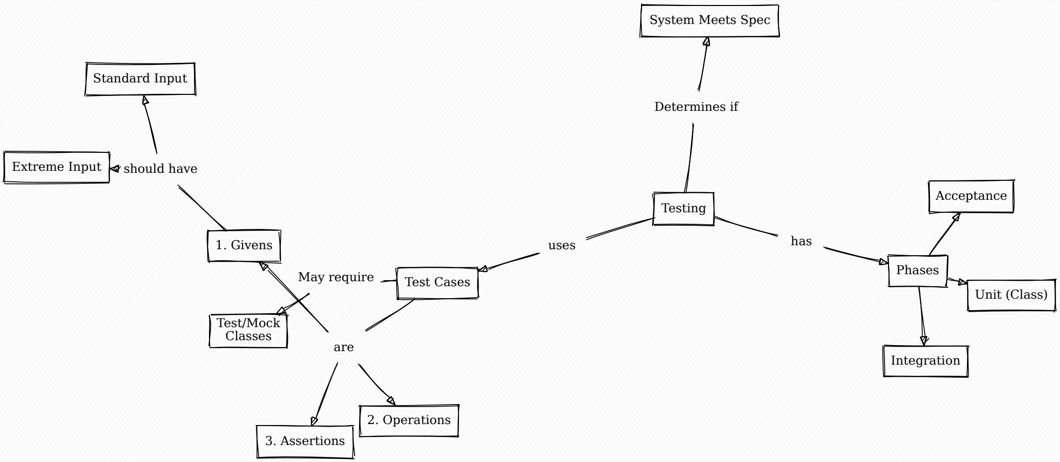
Other Types of Tests

- All testing is about transforming state and asserting something happens
- Most developers expected to write unit tests
- Integration and Acceptance testing can use similar techniques
  - Sometimes expert "tester" role for these larger scale tests
  - Often need interaction from the system:
    - Simulating user input/Checking display output
    - Custom test databases
    - Harder, but not impossible, in unit tests

Non-Functional Tests

- Unit/Integration/Acceptance tests usually check the *functionality*
  - Does this class/component/system meet the specification
- You might need to check non-functional requirements
  - "Can we handle 10,000 requests per minute"?
  - Needs benchmarking harnesses
    - Sometimes specialised roles for this "Site Reliability Engineering"
  - Challenge: Simulating a "realistic enough" environment
    - Similar machine, similar workloads etc
    - E.g. 10,000 trivial requests is possible; what about complex ones?

What Did We Learn



Q&A From the Lecture

Q: could you explain what mock test is and how could I achieve mock manually

A: mocks are objects used within a test; not a test case themselves, e.g. a "mock test" does not make sense but a "test that uses a mock" does.

Mocks allow use of dependencies that would otherwise be difficult to work with, e.g. production databases, real billing services, or any other class that takes a lot of initialisation to use.

The next few slides have a longer example of database handling.

Q&A From the Lecture

We might want to test a user class can synchronise with database data:

```
public void testSyncData() {
  ProductionDatabase d = new ProductionDatabase();
  User u = new User();
  u.synchonise(d, 123 /* user id */)
  assert(u.isCustomer() == true;)
}
```

There's a few issues: 1. We can read possibly private data in the tests 2. We might accidentally write over the customer with user id 123! 3. Setting up/tearing down a Database for a single test case has overheads

Q&A From the Lecture

A solution is to use a **mock**, a class that acts *like* a database, e.g. has the same interface, but does not use a real database.

To ensure we have the same interface we either introduce a new `interface` type, or sub-class:

```java
public class TestDatabase extends ProductionDatabase {
    public void syncUser(User u, int userId) {
        // Test class so it's okay to match on specific id's
        if (userId == 123) {
            u.setCustomer();
        }
    }
}
```

Q&A From the Lecture

This `syncUser` method is much simpler than one that reads from a real database (which would probably need to generate SQL queries etc).

We can then tweak our test case:

```
public void testSyncData() {
  ProductionDatabase d = new TestDatabase();
  User u = new User();
  u.synchonise(d, 123 /* user id */)
  assert(u.isCustomer() == true;)
}
```

The important bit is that we aren't testing the *database*, we are testing the *user* class that just so happens to need a database, so we give a fake one.