Testing doubles

Gal Lalouche
Oath Inc.

Computer Science
Technion

Spring 2018
Motivation

- Initializing collaborators can include a lot of boilerplate, which is
  - Irrelevant and noisy at best
  - Error-prone at worst
- There may be bugs in the dependencies
  - Or dependencies may not even be written yet!
- I/O operations (databases/files/sockets) in test subjects slow down tests
  - They might have nasty side effects or need to be cleaned up after each test
  - Or our code may have side effects that we want to verify, such as making sure a remote server was updated
- Using concrete classes couples our tests with those classes
Solution: Replace dependencies with test-specific objects

- Simple dependencies that are 100% bug free*
  - *Although bugs may appear in the mock objects configurations
- Easy to control their behavior
- Easy to verify their use by the testing class
- These are called Test doubles
According to Martin Fowler, there are several types of test doubles:

1. **Dummies** are passed around but never actually used
   - Usually they are just used to fill parameter lists
   - Should only be used for external APIs, because why would you require useless parameters in your API?

2. **Stubs** provide canned answers to calls made during the test, if at all
   - Used for stuff we usually don’t care to test, such as logging systems

3. **Fakes** actually have working implementations, but usually take some shortcut making them unsuitable for production
   - An in memory database is a classical example: much faster and doesn’t need to setup and cleaned-up
**Spies** are doubles that also record information on how they were called.

**Mocks** are pre-programmed with expectations which form a specification of the calls they are expected to receive.

In the tutorial and homework, you will work with a framework called **Mockito**, which uses slightly different terminology:

- Mocks are verified after the invocation, not before.
- Spies wrap real objects and allow to configure and inspect them.
Which double should we use?

Principle of Least Power (PoLP)

Given a choice of solutions, pick the **least powerful solution** capable of solving your problem

According to the principle, you should use the **first** (i.e., weakest) item that solves your problem

- No need to use doubles? Then don’t!
- Can get by with a stub? Use that
- Want more control? Use a fake
- Need to control and verify **everything**? Go with mocks
Applying PoLP has several advantages:

- **Declarative.** We’re specific in our minimalist needs
- **Safer.** The less power we have (that we don’t need), the fewer errors we’ll make
  - When using test doubles, test failures are more commonly caused by bad configuration than actual production bugs
- **Fewest Elements.**
- **Resistance to change.** It’s easier to change less powerful constructs
In this lecture we will focus on three kinds of test doubles

1. Stubs
2. Fakes
3. Mocks
Stubs

Use a stub when you just need dead simple functionality

```java
public class EmailSender {
    private final EmailServer server;
    public EmailSender(EmailServer server) { this.server = server; }
    /** @return true iff send was successful. */
    public boolean sendEmail(Email e) {
        try {
            server.send(e);
            return true;
        } catch (Exception e) {
            return false;
        }
    }
}

@Test public void sendEmailReturnsFalseOnException() {
    EmailSender sender =
        new EmailSender(e -> { throw new RuntimeException(); });
    assertFalse-send.send(new Email());
}
```
Fakes are alternative implementations that should only be used in tests

- Support all the same functionality as the production class its imitating
- But don’t suffer from the same limitations
- No disk or network access
- No other side effects
- Easier to set up than real objects since there are no transitive dependencies
- Can define exact behavior for testing

Remember, production code is general, testing code is concrete
public class TopStudentExtractor {
    private final Database db;
    public TopStudentExtractor(Database db) { this.db = db; }
    public void addStudent(Student s) {
        db.executeQuery("INSERT INTO students (...) ");
    }
    public String getStudentNameWithTopGrade() {
        return db.executeQuery("SELECT NAME FROM students ORDER BY ...");
    }
}

@Test public void test() {
    TopStudentExtractor sd = new TopStudentExtractor(new MemoryDB());
    sd.addStudent(new Student("Tom", 90));
    sd.addStudent(new Student("Dick", 80));
    sd.addStudent(new Student("Harry", 95));
    assertEquals("Harry", sd.getStudentNameWithTopGrade());
}
Fakes (cont.)

Fakes often expose API allowing for easy configuration of their state

```java
public class WeekendChecker {
    private final Clock clock;
    public WeekendChecker(Clock clock) {
        this.clock = clock;
    }
    public boolean isWeekend() {
        switch (clock.getDay()) {
            case Day.FRIDAY:
            case Day.SATURDAY:
                return true;
            default:
                return false;
        }
    }
}

@Test public void testWeekend() {
    // FakeClock extends/implements Clock
    FakeClock clock = new FakeClock();
    clock.setDay(Day.SATURDAY);
    WeekendChecker checker = new WeekendChecker(clock);
    assertTrue(checker.isWeekend());
}
```

Mocks are fakes that can be queried on their interaction history

- The most powerful test double variant
- Can be set up just as fakes
- Keep history of all invocation and parameters passed
- Usually created with frameworks like Mockito
  - Manual mock creation is a pain
  - By using reflection we can generate type-safe mocks for all types
Use mocks when you want to verification interactions with dependencies

```java
public class Checkout {
    private final PaymentProcessor ps;
    public EmailSender(PaymentProcessor ps) { this.ps = ps; }
    public void checkout(Cart cart) {
        int total = cart.items().stream().mapToInt(Item::price).sum();
        ps.charge(total);
    }
}
```

```java
@Test
public void sendsMailToAllRecipients() {
    // extends/implements PaymentProcessor
    MockPaymentProcessor processor = new MockPaymentProcessor();
    Checkout co = new Checkout(processor);
    Cart cart = new Cart();
cart.add(new Item(10));
cart.add(new Item(20));
cart.add(new Item(30));

    co.checkout(cart);

    assertEquals(60, processor.getChargedAmount())
}
```
Configuration vs. Verification

Prefer configuration to verification of test doubles

```java
class StringJoiner {
    public String static join(
        Supplier<String> s1, Supplier<String> s2) {
            return s1.get() + s2.get().reverse();
        }
}

@Test public void configurationTest() {
    assertEquals("foorab", StringJoiner.join(() -> "foo", () -> "bar"));
}

@Test public void verificationTest() {
    MockSupplier ms1 = new MockSupplier();
    MockSupplier ms2 = new MockSupplier();
    StringJoiner.join(ms1, ms2);
    assertEquals(1, ms1.getCount());
    assertEquals(1, ms2.getCount());
}
```
Applies the principle of least power

Less coupled to the implementation

Only verify interaction if it’s an integral part of the class’s definition
  For example, if you want to minimize expensive interactions

 Doesn’t actually check the code works, just what steps it did in the process

But definitely avoid mocks that do both!
  Usually Redudant, as in the above example
  You want your data to go one way only
  Split class into two separate interfaces

See Google’s testing blog entry on the subject
class EmailMaker {
    public static Email make(String name, String msg, EmailService service) {
        String address = service.getName(name);
        return new Email(address, msg);
    }
}

@Test public void redundantTest() {
    MockService mock = new MockService();
    mock.put("someone", "foo@bar.com");

    assertEquals("foo@bar.com", EmailMaker.make("someone", "hello", mock).getAddress());
    // Already covered by the above assertion!
    assertEquals("someone", mock.getAskedNames());
}
Refactoring configuration with verification

class EmailSenderBad {
    public static sendEmail(String msg, String name, EmailService service) {
        String address = service.getAddressForName(name);
        service.send(msg, address);
    }
}

@Test public void testBad() {
    EmailServiceMock mock = new EmailServiceMock();
    mock.setAddress("someone", "foo@bar.com");

    EmailSenderBad.sendEmail("hello", "someone", mock);

    assertEquals("hello", mock.getMessagesTo("foo@bar.com"));
}
```java
class EmailSenderGood {
    public static sendEmail(String msg, String name, NameService nameService, SendService sendService) {
        String address = nameService.getAddressForName(name);
        sendService.send(msg, address);
    }
}

@Test public void testGood() {
    MockNameService nameService = new MockNameService();
    mock.put("someone", "foo@bar.com");
    MockSenderService senderService = new MockSenderService();
    EmailSenderBad.sendEmail("hello", "someone", nameService, sendService);
    assertEquals("hello", mock.getMessagesTo("foo@bar.com"));
}
```
Testing doubles as a code smell

As we saw, testing doubles are a very powerful tool in our arsenal

- They’re much easier to set up since they usually don’t have dependencies of their own
- They let us avoid side effects we don’t want in our tests
- They let us keep our unit tests truly “unity”
- We get fine-grained control on our class dependencies
  - We can return values
  - Throw exceptions
  - And verify what methods were called

But there are negatives to consider…
There are several issues that arise from the (over)use of testing doubles in general, and mocks in particular

1. Code echoing
2. Side effect enabling
3. White-boxing and implementation coupling
Overusing mocks can often lead to tests that just mirror production code

```java
public class UserManager {
    private final Database db;
    public UserManager(Database db) { this.db = db; }
    public void removeUser(String id) {
        db.executeQuery("DELETE FROM users WHERE id=\" + id);
    }
}

@Test public void testRemoveUser() {
    MockDb db = new MockDb();
    UserManager sd = new UserManager(mockDb);
    sd.removeUser("foo");
    assertEquals("DELETE FROM users WHERE id=foo", db.getLastQuery());
}
```

This test is pointless and adds nothing of value
Side effect enabling

- Mocks make it relatively easy to test for side effects
  - Just define a mock, and check interactions with it
- But you should generally try to avoid side effects
  - They make it harder to reason about your code
  - They make it harder to reuse your code
  - And, yes, they do make your code harder to test thoroughly
  - We will talk about this at length when we discuss functional programming later in the course
White-boxing

Perhaps the most striking consequence of test doubles is enforcing a strictly white-box testing methodology:

- This is practically by definition
- Tests are very coupled with the actual implementation
  - Dependencies change? Tests break
- Brittle and tight coupling in turn mean that our tests are no longer independent of our product code
- Worse, Refactoring production code is now harder, since we have to change our tests as well
  - This negates the (major) point of writing tests to begin with!
- Too many unit tests and not enough (real) integration tests will eventually result in bugs
  - Test doubles are integration tests to some degree, but they don’t bestow the same amount of confidence
Code smell summary

But, sometimes, test doubles are the right choice

- You can’t do network access in your unit tests
- You don’t want to do IO or deal with persistent memory
- You want isolated unit tests, so bugs are easier to squash
Writing tests has substantial costs attached

- Writing tests often takes more time than writing the tested code itself, and almost always produces more lines of code.
- But without tests, we can’t refactor; if we can’t refactor, our code quality will deteriorate over time.
- But not all tests are created equal.
  - Some tests are harder to write than others.
  - Some tests are more beneficial than others.
- This section based on Selective Unit Testing—Costs and Benefits.
Cost of tests (cont.)

We’ll look at two main axes

1. Cost of test
2. Benefit of tests
Generally speaking, classes with more dependencies are more difficult / annoying to test

- Such classes are often called coordinators or managers
- The more dependencies, the more test doubles need to be configured
  - Taking more time, and error-prone
- More important, the more external you are coupled to, the more likely test refactoring will be necessary to accommodate for changes
  - These are an absolute time waste
  - But you can’t skip them, otherwise your tests won’t compile / will fail
We already discussed testing benefits in the previous lecture, but what code benefits especially from having tests?

- Complicate code,
  - Input-output example tests help document behavior
- Code that is very likely to have multiple internal refactorings
  - That is, not ones caused by dependency changes
- Code with a lot of edge-cases
Cost-benefits diagram

Test cost

Test benefits

Algorithmic code (II)

Complicated code (IV)

Trivial code (I)

Coordinators and Managers (III)
Diagram analysis

- **Trivial code** has neither dependencies nor real logic of its own; no need to test
- **Algorithmic code** has few dependencies that need to be replaced, but benefits greatly from tests; test this!
- **Coordinators and Managers** have a lot of dependencies, but not a lot of complicated tests; can avoid testing
- **Complicated code** has both dependencies and logic, usually calling for refactoring
Of course, the above diagram is a gross \textit{simplification}

- Separation between algorithms and coordinators is a nice guideline, but \textit{perfect dichotomy} is neither achievable nor desirable
  - It’s okay to be do a little bit logic and a little bit of coordination is the same class
  - Most classes will
  - It’s when the class does both heavily that you should start worrying

- Pretty much any non-trivial test is useful to some degree
  - Increases confidence
  - Documentation

- It’s impossible to know what changes will be needed in the future
  - Or what better design we might come up with next

- The more tests we have, the easier it is to refactor safely
  - And vice versa: without tests, you can’t refactor
  - And virtually all code will be refactored, even coordinators
Summary

- Apply the principle of least power
- Don’t mock if you don’t have to mock
  - You don’t need to mock fast dependencies without side effects
  - Dependable dependencies, e.g., from the standard library
- If possible, refactor your code so test doubles aren’t as necessary