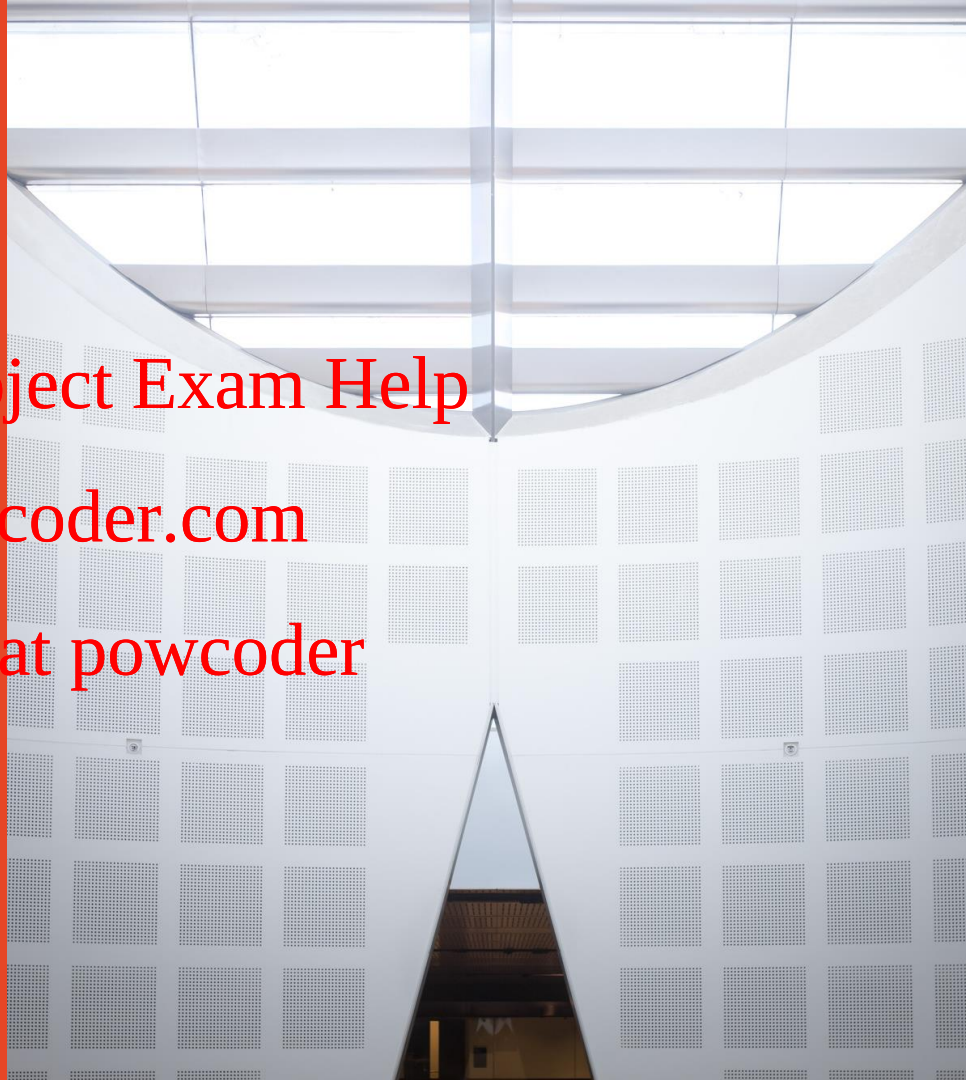
Software Design and Construction 2
SOFT3202 / COMP9202

Advanced Testing Techniques (1)

Dr. Basem Suleiman

School of Information Technologies

THE UNIVERSITY OF SYDNEY



Copyright Warning

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

Agenda

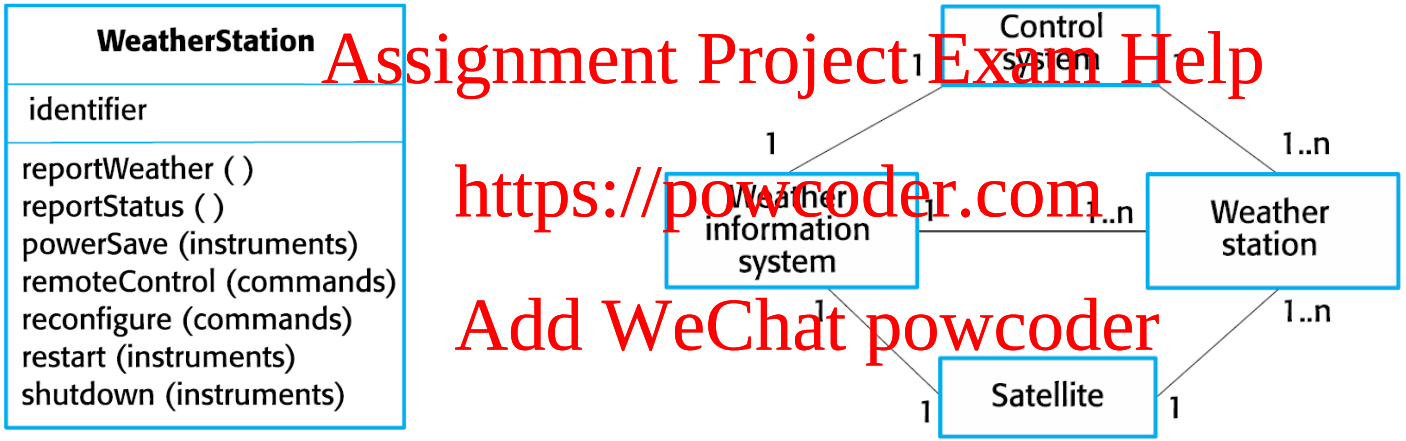
- Testing Types
  - Integration Testing, Regression Testing
- Advanced Testing Techniques
  - Test doubles (Dummies, Fakes, Stubs, Spies, Mocks)
  - Contract Test
- Testing Frameworks
  - Mockito

Advanced Testing Types

Integration testing, regression testing

THE UNIVERSITY OF SYDNEY

Software Components/Sub-systems



Integration Testing

- The process of verifying interactions/communications among software components behave according to its specifications
- Independently developed (and tested) units may not behave correctly when they interact with each other
- Activate corresponding components and run high-level tests
- Incremental integration testing vs. “Big Bang” testing

Interaction Errors

- Parameter interfaces
  - Methods in objects have a parameter interface
- Procedural interfaces
  - Objects and reusable components
- Message passing interfaces
  - One component encapsulates a service from another component by passing a message to it
- Shared memory interfaces
  - Interfaces in which block of memory is shared between components (e.g., embedded systems)

Your Testing Exposed Bugs

- What would you do when your testing reveal bugs/errors?
- You fixed the discovered bugs, what should happen next?
- You extended one class with additional functionality (new feature), what should happen next?

.

Regression Testing

- Verifies that the software behaviour has not changed by incremental changes to the software
  - Bug fixes, code extension, code enhancements
- Modern software development processes are iterative/incremental
- Changes may be introduced which may affect the validity of previous tests
- Regression testing is to verify
  - Pre-tested functionality (and non-functional properties) still working as expected
  - No new bugs are introduced

Regression Testing – Techniques

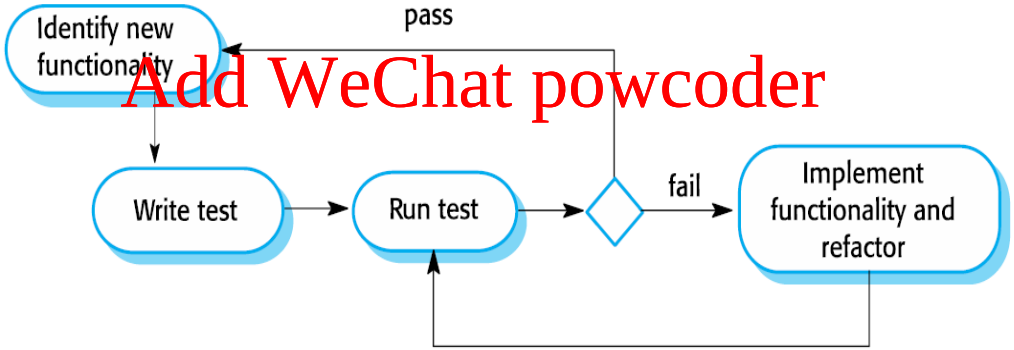
Type	Description
Retest All	Re-run all the test cases in a test suit
Test Selection	Re-run certain test cases based on the changes in the code
Test case prioritization	Re-run test cases in order of its priority; high, medium, low. Priority determined by how criticality and impact of test cases on the product
Hybrid	Re-run selected test cases based on it's priority

. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.460.5875&rep=rep1&type=pdf

Test-Driven Development

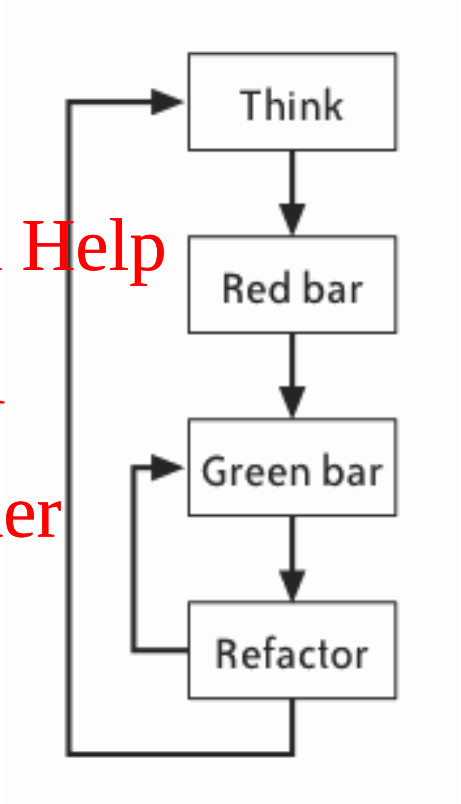
Test-driven Development

- A software development approach for developing the code incrementally along with a set of tests for that increment
  - Write tests before code
- All tests must pass before starting the next increment
- Introduced in the XP agile development method



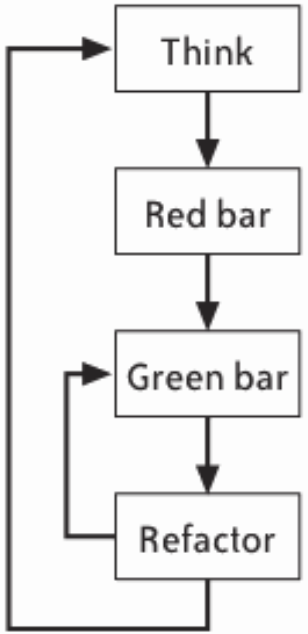
TDD Cycle

- "A rapid cycle of testing, coding, and refactoring"
  – Kent Beck
- "Every few minutes, TDD provides proven code that has been tested, designed, and coded"
- **Red, Green, Refactor cycle**



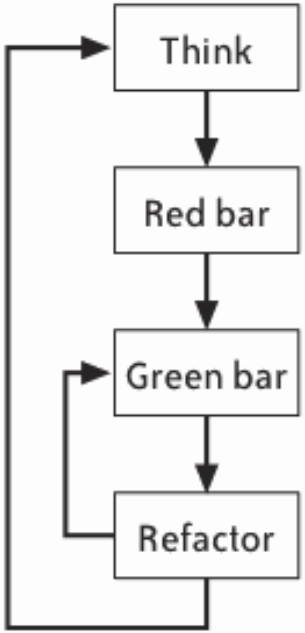
TDD Cycle – Think

- **Think** of a behavior you want your code to have (small increment; few lines of code)
- **Think** of a test (few lines of code) that will fail unless the behavior is present
- Pair programming helps
  - Driver and navigator



TDD Cycle – Red (Run the Test)

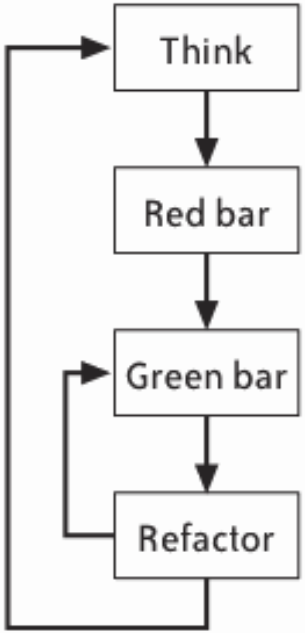
- Write the tests – only enough code for the current increment of behavior
  - Typically less than 5 lines of code
- Code for the class behavior and its public interface (encapsulation)
  - Tests use method and class names do not exist yet
- Run your entire suite of tests and enjoy the test failure
- Results in Red progress bar (testing tools)



https://www.jamesshore.com/Agile-Book/test_driven_development.html

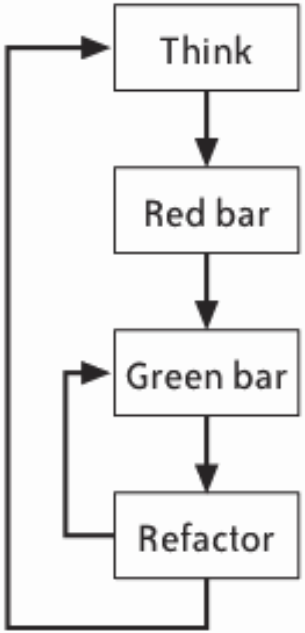
TDD Cycle – Green (Write Code)

- Write the code; just enough to get the test to pass
  - Less than 5 lines
  - It's okay to hard code, you'll refactor
- Run your tests again, and enjoy the tests passing
- Results in Green progress bar (testing tools)



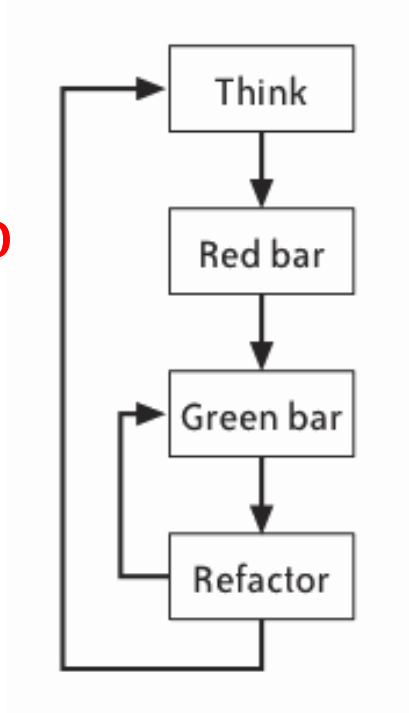
TDD Cycle – Refactor

- Review the code and look for possible improvements
  - Ask your navigator if they made any notes
- Series of very small refactorings
  - 1-2 minutes each, no longer than 5 minutes
- Run the tests after each refactoring
  - Should always be green (pass)
  - Test failed and no obvious answer, get back to good code
- Refactor many times, improve design
  - Refactoring isn't about changing behavior



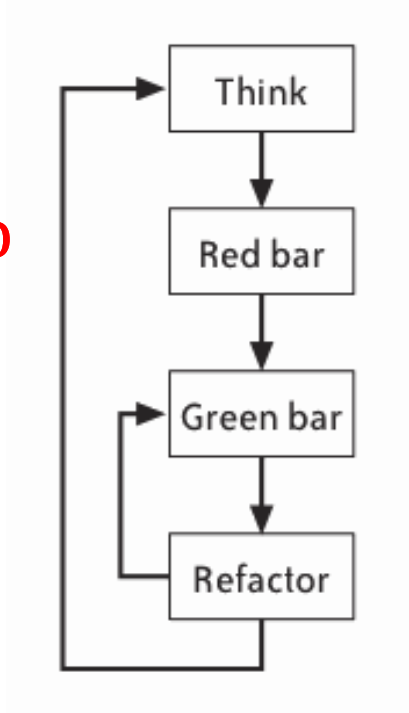
TDD Cycle – Repeat

- Repeat to add new behavior, start the cycle over again
- Tiny bit of well-tested, well-designed code will be incrementally created
- Typically, run through several cycles very quickly, then spend more time on refactoring
- Do not skip any step, especially refactoring, to speed up!



TDD – Red, Green, Refactor

- Use Red, Green, Refactor cycle to impalement TDD
- **Think** of a code behavior, then choose a small increment and then a test
- Write the test for the current increment code and run the entire suit of tests – should fail (**Red bar**)
- Write just enough code to get the test pass and run the tests again – should pass (**Green bar**)
- Review the code and look for improvements – small set of refactoring and run the tests after each (**Refactor**)



https://www.jamesshore.com/Agile-Book/test_driven_development.html

TDD – Example

- Java class to parse HTTP query string (name-value pair)
  - E.g., http://example.com/page/to/page?title=Central+Park&where=US
- Think
  - "class to separate name/value pairs into a HashMap" or "class to put one name/value pair into a HashMap"? Why?
  - Class *QueryString* won't return a HashMap, but a method (valueFor(name)) to access the name-value pairs. Shall you proceed with writing the test?
  - Count() method instead to return total number of name-value pairs (more suitable for one increment)
  -

https://www.jamesshore.com/Agile-Book/test_driven_development.html

TDD Example – Red Bar

```
public void testOneNameValuePair() {
    QueryString qs = new QueryString("name=value");
    assertEquals(1, qs.count());
}

public class QueryString {
    public QueryString(String queryString) {}
    public int count() { return 0; }
}
```

TDD Example – Green Bar & Refractor

```
public int count() { return 1; }
```


Refactor

Change the QueryString name to HttpQuery() – noted for next cycle

Another test to try

TDD Example – Repeat

Thinking

- Remove the hard-coded line but not time yet to deal with multiple query string
- Testing an empty string would require coding the count() properly


```
public void testNoNameValuePairs() {
    QueryString qs = new QueryString("");
    assertEquals(0, qs.count());
}
```


Emerging thoughts (noted for later cycles)

- Test the case of a null argument to the QueryString constructor
- Deal with the tests duplication tests that needed refactoring

TDD Example – Green & Refactor

```
public class QueryString {
    private String _query

    public QueryString(string queryString) {
        _query = queryString;
    }
    public int count() {
        if ("".equals(_query)) return 0;
        else return 1;
    }
}
```

Refactor (notes):

- Rename QuerySting
- **testNull()**
- Refactor duplicate tests

TDD Example – testNull()

- Test the case when the query string is null
- Red Bar – think of the behavior when the value is null
- Through an exception (Null is illegal) – simple design


```
public void testNull() {
    try {
        QueryString qs = new QueryString(null);
        fail("Should throw exception");
    }
    catch (NullPointerException e) {
        // expected
    }
}
```

```
public QueryString(String queryString) {
    if (queryString == null) throw new
    NullPointerException();

    _query = queryString;
}
```

TDD Example – valueFor()

- Implement valueFor() method to return the associated value give a name/value pair
- Emerging thoughts: test for a name doesn't exist (noted)


```
public void testOneNameValuePair() {
    QueryString qs = new
QueryString("name=value");
    assertEquals(1, qs.count());
    assertEquals("value",
qs.valueFor("name"));
 }
```



```
public String valueFor(String name) {
    String[] nameAndValue =
_query.split("=");
    return nameAndValue[1];
}
```

TDD Example – Repeat

- Code passed the tests, but it was incomplete
- Multiple name/value pairs …
- Repeat …

TDD – Benefits

- Help developers to understand the requirements and write better code
- Simplify debugging
  - Easier to find and fix mistakes in small code chunks
- Reduce cost of regression testing
- Improved design and code quality
  - Research shows TDD substantially reduces the incidence of defects
- Reuse tests as the software grow, and use it as documentation

Refactoring

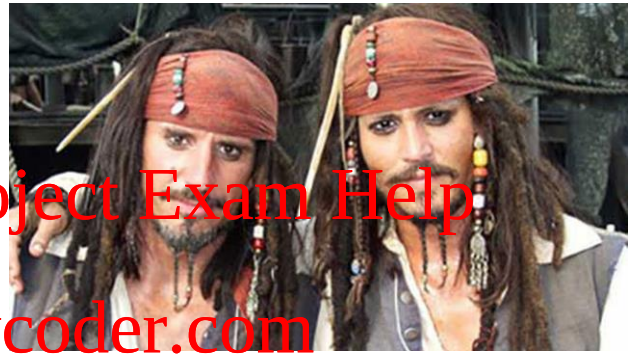
- "*Refactoring* is the process of changing the design of your code without changing its behavior" – Kent Beck
- Change the *how* not the *what*
- Refactoring is reversible!
- Analyze the design of existing code and improve it
- Code improvements can be identified with *code smells*

How to Refactor

- Refactor constantly in a series of small transformations
- Learn from in-depth catalog of refactoring
- Refactor intuitively through learning the mindset behind refactoring
- Learn how to refactor manually
  - Development frameworks/tools can help automating some refactoring
- Use continuous integration practices and automation tools
  - Version control system, build and test automation, IDEs

Test Double

Movie – "Stunt Double"



https://i.ytimg.com/vi/xm7kzxXHF38/maxresdefault.jpg
http://cdn.kickvick.com/wp-content/uploads/2014/07/celebrity-stunt-doubles.jpg
https://amp.businessinsider.com/images/525328ce6bb3f78e7afdcbb2-750-563.jpg

Test Double

- "*A **test double** is an object that can stand in for a real object in a **test**, similar to how a **stunt double** stands in for an actor in a movie*" – *Google Testing Blog*

  - Includes stubs, mocks and fakes
  - Commonly referred to as "mocks", but they have different uses!

- Why test double?
  - Dependency on components that cannot be used
  - Reduce complexity, test indecently

https://testing.googleblog.com/2013/07/testing-on-toilet-know-your-test-doubles.html

Test Double – Types

Type	Description
Dummy	Pass object(s) that never actually used (to fill parameter list)
Stub	Test-specific object(s) that provide indirect inputs into SUT
Spy	Capture indirect output calls made by the SUT to another component for later verification
Fake	Objects to provide simpler implementation of a heavy component
Mock	Object(s) that verify indirect output of the tested code

Dummy Object

- Dummy, dummy parameter/value
- Pass object with no implementation (dummy) and never actually used
  - E.g., Fill in parameter lists
- SUT's methods to be called often take objects stored in instance variables
  - Those objects, or some of its attributes, will never be used in the testing
- Preparing the SUT into right state (conform to the signature of some methods need to be called)

Dummy Object – Example

```
2  public void testInvoice_addLineItem_noECS() {
3        final int QUANTITY = 1;
4        Product product = new Product(getUniqueNumberAsString(), getUniqueNumber());
5        State state = new State("West Dakota", "WD");
6        City city = new City("Centreville", state);
7        Address address = new Address("123 Blake St.", city, "12345");
8        Customer customer = new Customer(getUniqueNumberAsString(), getUniqueNumberAsString(),
9                                         address);
10       Invoice inv = new Invoice(customer);
11       // Exercise
12       inv.addItemQuantity(product, QUANTITY);
13       // Verify
14       List lineItems = inv.getLineItems();
15       assertEquals("number of items", lineItems.size(), 1);
16       LineItem actual = (LineItem)lineItems.get(0);
17       LineItem expItem = new LineItem(inv, product, QUANTITY);
18       assertLineItemsEqual("",expItem, actual);
19   }
```

(Test) Stub

- A test-specific object that provides indirect inputs during tests
  - E.g., Object requires data from a database to answer a method call
- Used to verify logic independently when it depends on inputs from other components
- Verify indirect inputs of the SUT
- It does not deal with indirect outputs of the system

(Test) Stub – Example

```
2  public void testDisplayCurrentTime_AtMidnight() throws Exception {
3        // Fixture setup:
4        //      Test Double configuration
5        TimeProviderTestStub tpStub = new TimeProviderTestStub();
6        tpStub.setHours(0);
7        tpStub.setMinutes(0);
8        //   Instantiate SUT:
9        TimeDisplay sut = new TimeDisplay();
10       //      Test Double installation
11       sut.setTimeProvider(tpStub);
12       // exercise sut
13       String result = sut.getCurrentTimeAsHtmlFragment();
14       // verify outcome
15       String expectedTimeString = "<span class=\"tinyBoldText\">Midnight</span>";
16       assertEquals("Midnight", expectedTimeString, result);
17   }
```

(Test) Spy

- Capture output calls made by the SUT to another component for later verification
  - Verify indirect outputs of the SUT
- Get enough visibility of the outputs generated by the SUT (observation point)
- E.g., email service that records no. of messages sent

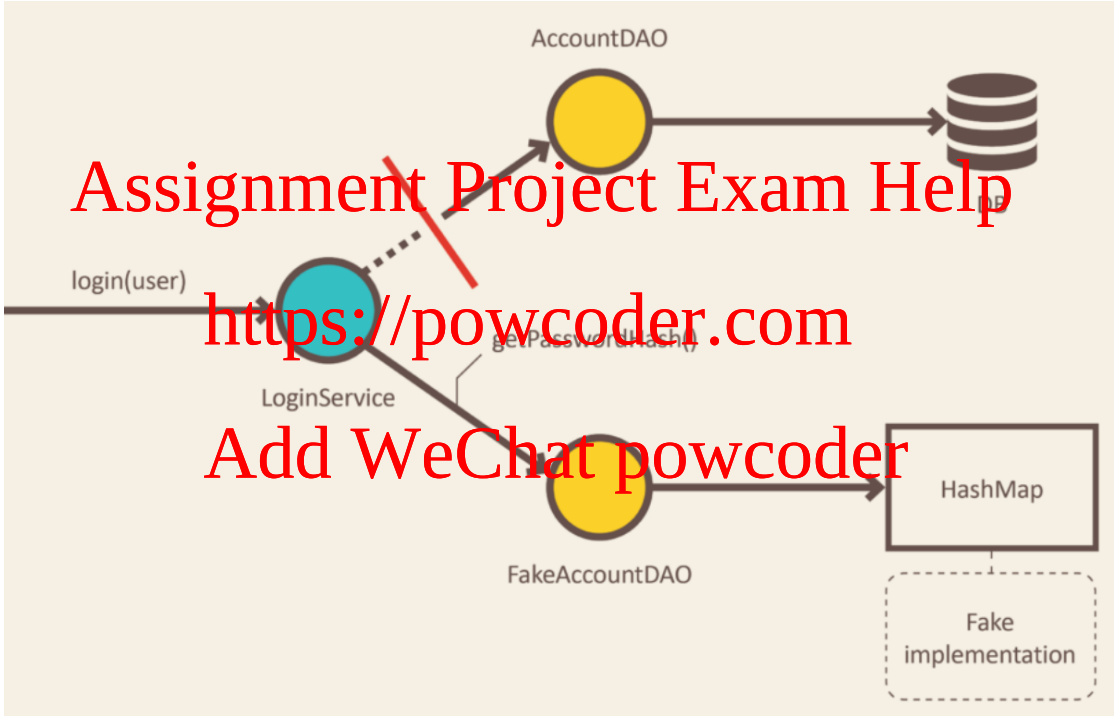
(Test) Spy

- Capture output calls made by the SUT to another component for later verification
  - Verify indirect outputs of the SUT
- Get enough visibility of the outputs generated by the SUT (observation point)
- E.g., email service that records no. of messages sent

Fake (Object)

- Objects to provide simplified implementation of a heavy (real) component
  - E.g., in-memory implementation of repository using simple collection to store data
- SUT depends on other components that are unavailable or make testing complex or slow
  - Run tests faster
- Should not be used when want to control inputs to SUT or outputs of SUT

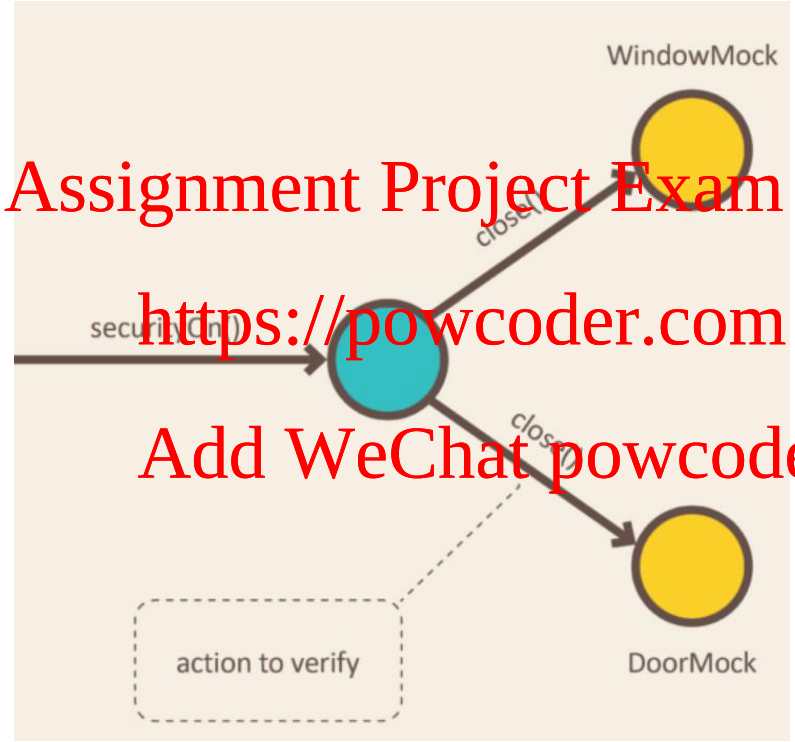
Fake (Object) – Example



Mock (Object)

- Object(s) that verify indirect output of the SUT
  - E.g., function that calls email sending service, not to really send emails but to verify that email sending service was called
- Calling real implementation during testing is tedious, or the side effect is not the testing goal
- Unlike all doubles, mocks verify correctness against expectations

Mock (Object) – Example



Mock (Object)

```
1
2  public class SecurityCentral {
3      private final Window window;
4      private final Door door;
5
6      public SecurityCentral(Window window, Door door) {
7          this.window = window;
8          this.door = door;
9      }
10
11     void securityOn() {
12         window.close();
13         door.close();
14     }
15 }
```

```
1
2  public class SecurityCentralTest {
3      Window windowMock = mock(Window.class);
4      Door doorMock = mock(Door.class);
5
6      @Test
7      public void enabling_security_locks_windows_and_doors() {
8          SecurityCentral securityCentral = new SecurityCentral(windowMock, doorMock);
9          securityCentral.securityOn();
10         verify(doorMock).close();
11         verify(windowMock).close();
12     }
13 }
```

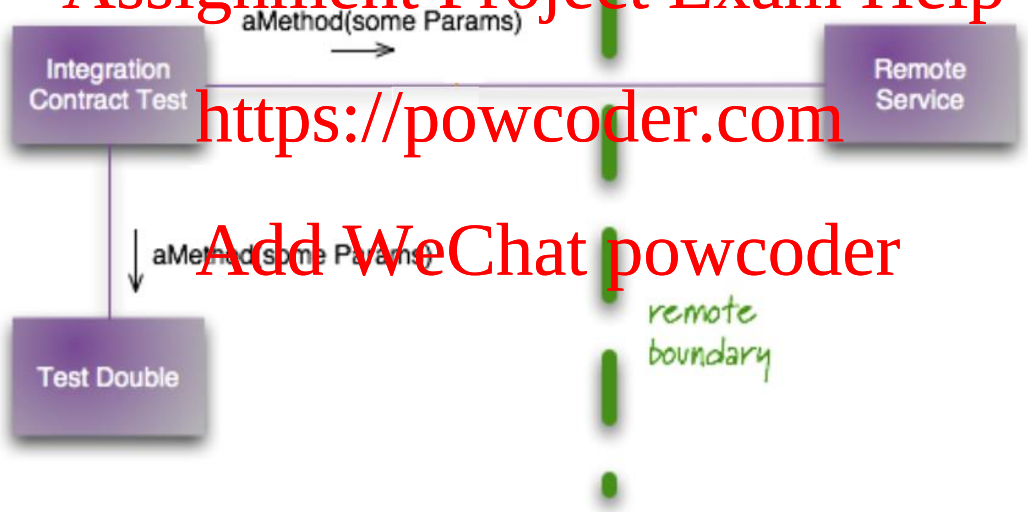
Test Doubles

- Understand the differences carefully and use the one that serve the verification type and purpose and how it should be run
- Don't be fooled by the mocking frameworks terminology – focus on the verification purpose
- Read Fowler's Mocks aren't Stubs
- Check xUnit Test Patterns for more advanced details

Contract Test

Test Double – External Services

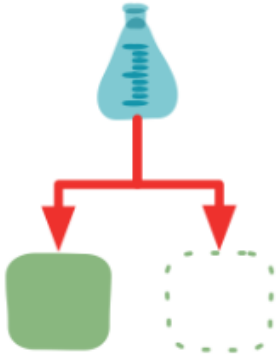
- Test double to interact with external/remote service
  - How accurate/reliable is a test double?



https://martinfowler.com/bliki/ContractTest.html

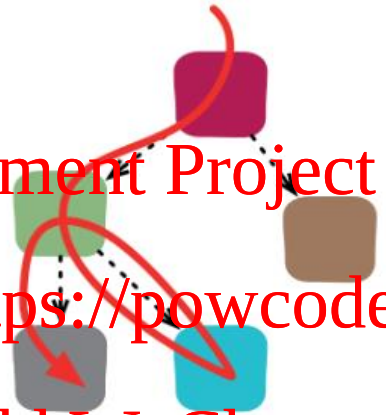
Contract Test

- The process of running periodic tests against real components to check the validity of test doubles results
- How?
  - Run your own test against the double
  - Periodically run separate contract tests (real tests to call the real service)
  - Compare the results
  - Check the test double in case of results inconsistency/failures
  - Also, consider service contract changes



https://martinfowler.com/bliki/ContractTest.html

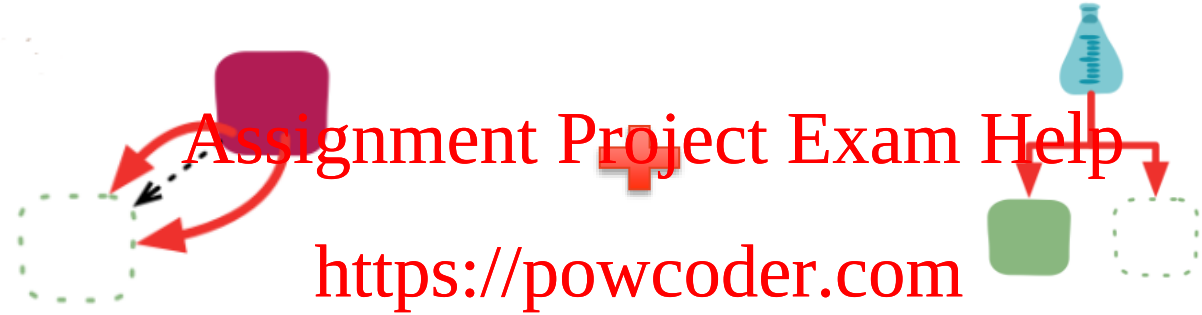
Integrated (Broad) Tests



"*Broad tests done with many modules active*" – ***integrated*** *testing*

Read more for further discussion - https://martinfowler.com/bliki/IntegrationTest.html

Collaborative (Narrow) Tests



"*Narrow tests of interaction with individual test doubles*" – **Collaboration Tests**

"supported by ***Contract Tests*** to ensure the faithfulness of the double" – Contract Tests

Read more for further discussion - https://martinfowler.com/bliki/IntegrationTest.html

Integration Testing Frameworks

Mockito

THE UNIVERSITY OF SYDNEY

Mocking Frameworks

- Mockito
- JMock
- EasyMock
- Mountebank
- Others …

http://www.mbtest.org/
http://jmock.org/
http://easymock.org/

Mockito

- An open source testing (test spy) framework for Java
  - It has a type called 'spy' which is partial mock[1]
- Verify interactions after executing tests (what you want)
  - Not expect-run-verify (look for irrelevant interactions)
  - Interaction among objects/components not unit testing
- Allows to specify order of verification (not all interactions)

https://github.com/mockito/mockito/wiki/FAQ

Mockito – Constructs

Mockito Features	Description
mock(), @Mock or Mokito.mock()	Different ways to create a mock
Answer or MockSettings	Interfaces to specify how a mock should behave (optional)
when()	Specify the mock to return a value when a method is called
Spy() or @Spy	Caution: creates a (partial mock) for a given object
@InjectMocks	Automatically inject mocks/spies annotated with @Mock() or @Spy()
verify()	Check methods were called with given arguments

Note: call MockitoAnnotations.initMocks(testClass) (usually in a @Before method) to get the annotations to work. Alternatively, use MockitoJUnit4Runner as a JUnit runner

http://static.javadoc.io/org.mockito/mockito-core/2.24.0/org/mockito/Mockito.html

Mockito Example

```java
public class GradesService {
    private final Gradebook gradebook;

    public GradesService(Gradebook gradebook) {
        this.gradebook = gradebook;
    }

    Double averageGrades(Student student) {
        return average(gradebook.gradesFor(student));
    }
}
```

```java
public class GradesServiceTest {
    private Student student;
    private Gradebook gradebook;

    @Before
    public void setUp() throws Exception {
        gradebook = mock(Gradebook.class);
        student = aNewStudent();
    }

    @Test
    public void calculatesGradesAverageFor_student() {
        when(gradebook.gradesFor(student)).thenReturn(grades(8, 6, 10)); //stubbing gradebook
        double averageGrades = new GradesService(gradebook).averageGrades(student);
        assertThat(averageGrades).isEqualTo(8.0);
    }
}
```

Mockito – Method Call

- Use Mockito.when() and thenRturn() to specify a behavior when a method is called
- Example of methods supported in Mockito

Method	Purpose
thenReturn(valueToBeReturned)	Return a given value
thenThrow(Throwable tobeThrown)	Throws given exception
Then(Answer answer)	User created code to answer

Mockito – 'When' Example

```
1
2  when(mock.someMethod()).thenReturn(10);
3
4   //you can use flexible argument matchers, e.g:
5   when(mock.someMethod(anyString())).thenReturn(10);
6
7   //setting exception to be thrown:
8   when(mock.someMethod("some arg")).thenThrow(new RuntimeException());
9
10  //you can set different behavior for consecutive method calls.
11  //Last stubbing (e.g: thenReturn("foo")) determines the behavior of further consecutive calls.
12  when(mock.someMethod("some arg"))
13    .thenThrow(new RuntimeException())
14    .thenReturn("foo");
15
16  //Alternative, shorter version for consecutive stubbing:
17  when(mock.someMethod("some arg"))
18    .thenReturn("one", "two");
19  //is the same as:
20  when(mock.someMethod("some arg"))
21    .thenReturn("one")
22    .thenReturn("two");
23
24  //shorter version for consecutive method calls throwing exceptions:
25  when(mock.someMethod("some arg"))
26    .thenThrow(new RuntimeException(), new NullPointerException();
```


http://static.javadoc.io/org.mockito/mockito-core/2.24.0/org/mockito/Mockito.html#when-T-

Mockito – Verifying Behavior

- *Mockito.verify (T mockTobeVerified, verificationMode mode)*
  - Verifies certain behavior happened at least once (default) – e.g., a method is called once
  - Different verification modes are available

Verification Mode	Description
Times(int wantedNoCalls)	Called exactly n times, default = 1
atMost(in maxNoOfCalls)	Called at most n times
atLeast(int minNoOfCalls)	Called at least n times
never()	Never called
Timeout (int milliseconds)	Interacted in a specified time range

Mockito – Verifying Behavior Example

```
1
2 verify(mock, times(5)).someMethod("was called five times");
3
4    verify(mock, atLeast(2)).someMethod("was called at least two times");
5
6    //you can use flexible argument matchers, e.g:
7    verify(mock, atLeastOnce()).someMethod(anyString());
8
```


- Default mode is times (1) which can be omitted
- Argument passed are compared suing equals() method

Mockito – Verifying Order of Calls

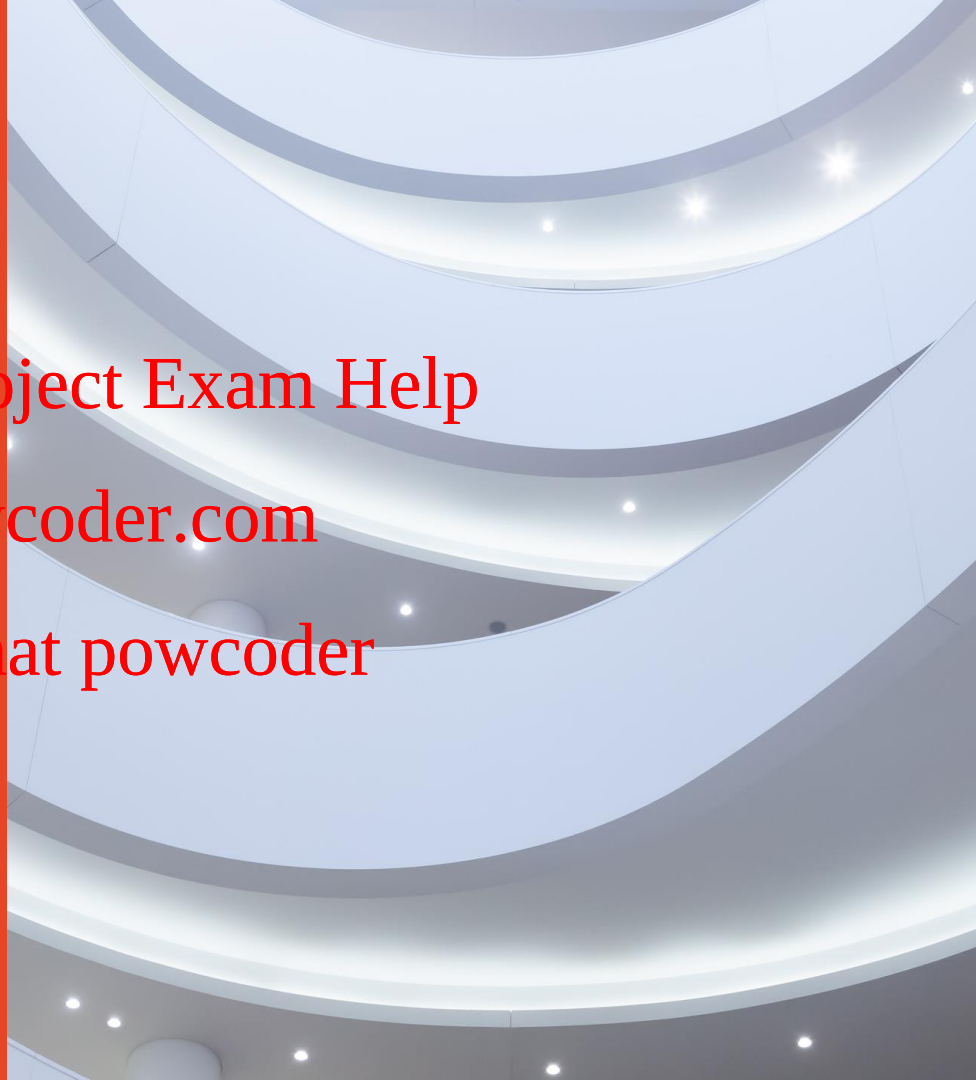
- InOrder (mocks) allows verifying mocks in order
  - *verify(mock)*: verifies interactions happened once in order
  - *verify(mock, VerificationMode mode)*: verifies interactions in order


```
1
2 InOrder inOrder = inOrder(firstMock, secondMock);
3
4  inOrder.verify(firstMock).add("was called first");
5  inOrder.verify(secondMock).add("was called second");
```

```
1
2 InOrder inOrder = inOrder(firstMock, secondMock);
3
4  inOrder.verify(firstMock, times(2)).someMethod("was called first two times");
5  inOrder.verify(secondMock, atLeastOnce()).someMethod("was called second at least once");
```

http://static.javadoc.io/org.mockito/mockito-core/2.24.0/org/mockito/InOrder.html

Writing Good Tests



THE UNIVERSITY OF SYDNEY

Writing Good Tests

- Reliable
  - Free of bugs, defects or errors
- Fast
  - Should not be counterproductive, will be run very frequently
- Keep it compact and readable
  - Lots of refactoring
  - Follow recommended coding practices (e.g., naming conventions, documentation)
- Cover wide range to show positive cases and errorenous code paths

https://github.com/mockito/mockito/wiki/How-to-write-good-tests

Writing Good Tests

- Do not mock everything
  - It's anti-pattern
- Understand mocking framework's capabilities
  - Mock syntax vs. actual purpose of mocking
  - Read Fowler's Mocks aren't Stubs
- Do not mock type you do not own
  - Third-party library or API – owner change the signature and behavior of the API
  - Contract test ?
- Do not mock value objects
  - Instantiating an object is too painful – not a valid reason
  - Can be a sign that the code needs some serious refactoring or use builders for the value objects (some tools such as Lombok can help)

https://github.com/mockito/mockito/wiki/How-to-write-good-tests

Next Lecture/Tutorial

W4 Lecture: Advanced Testing Techniques 2

W4 Tutorial + quiz

THE UNIVERSITY OF SYDNEY

References

- Ian Sommerville. 2016. Software Engineering (10th ed.) Global Edition. Pearson
- James Shore and Shane Warden. 2007. The Art of Agile Development
- Martin Fowler, various testing articles. https://martinfowler.com/
- Gerard Meszaros, xUnit Test Patterns: Refactoring Test Code. Addison-Wesley
- Martin Fowler, Mocks Arent Stubs, [https://martinfowler.com/articles/mocksArentStubs.html]
- Gaurav Duggal, Bharti Suri, Understanding Regression Testing Techniques. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.460.5875&rep=rep1&type=pdf
- Bernd Bruegge and Allen, H. Dutoit. 2009. Object-Oriented Software Engineering Using Uml, Patterns, and Java (3rd ed.). Pearson
- Michal Lipski, Pragmatists: Test doubles: Fakes, Mocks and Stubs. https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da