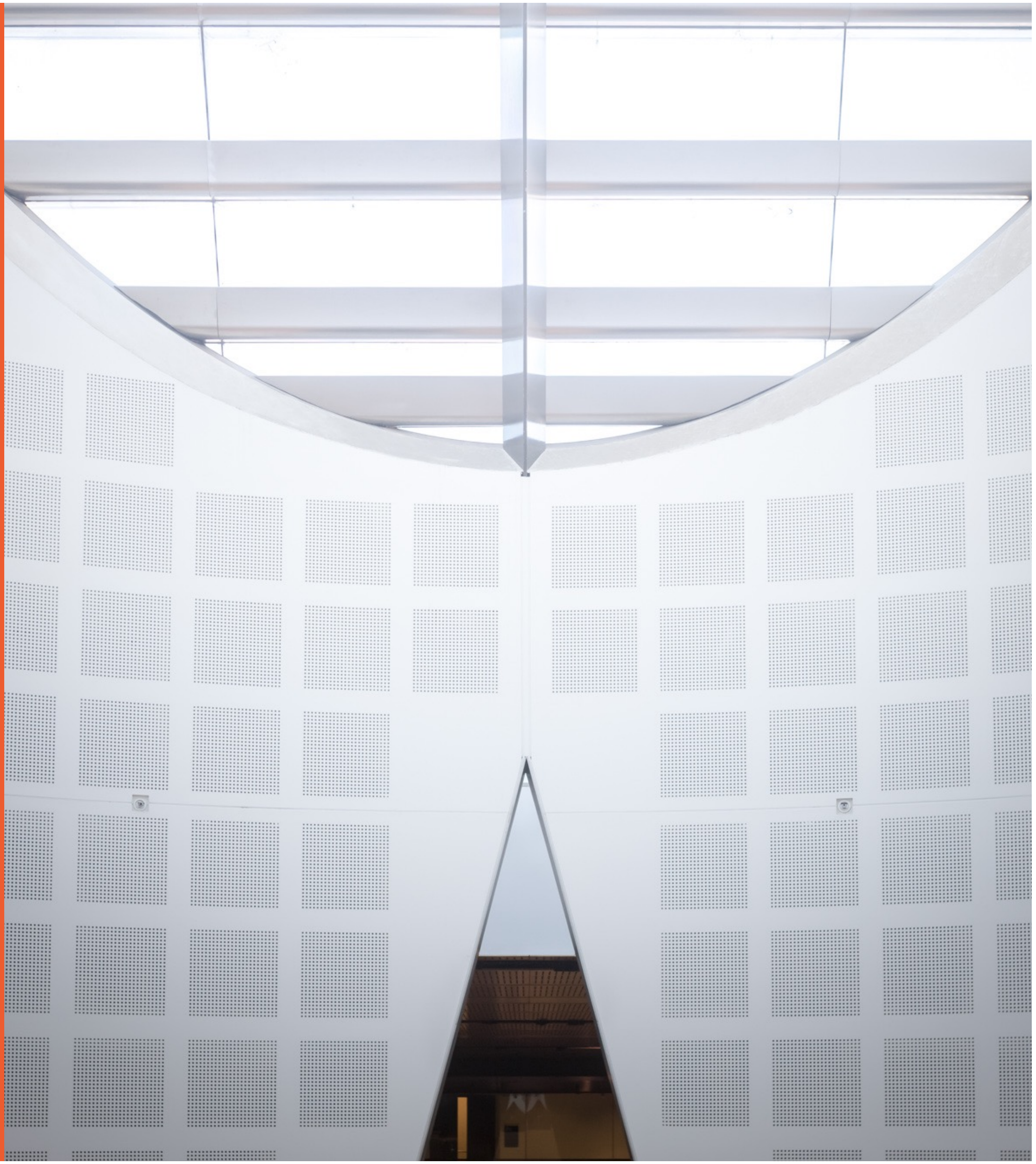
Software Design and Construction 2 SOFT3202 / COMP9202 Advanced Testing Techniques (1)

Prof Bernhard Scholz

School of Computer Science

THE UNIVERSITY OF SYDNEY



Agenda

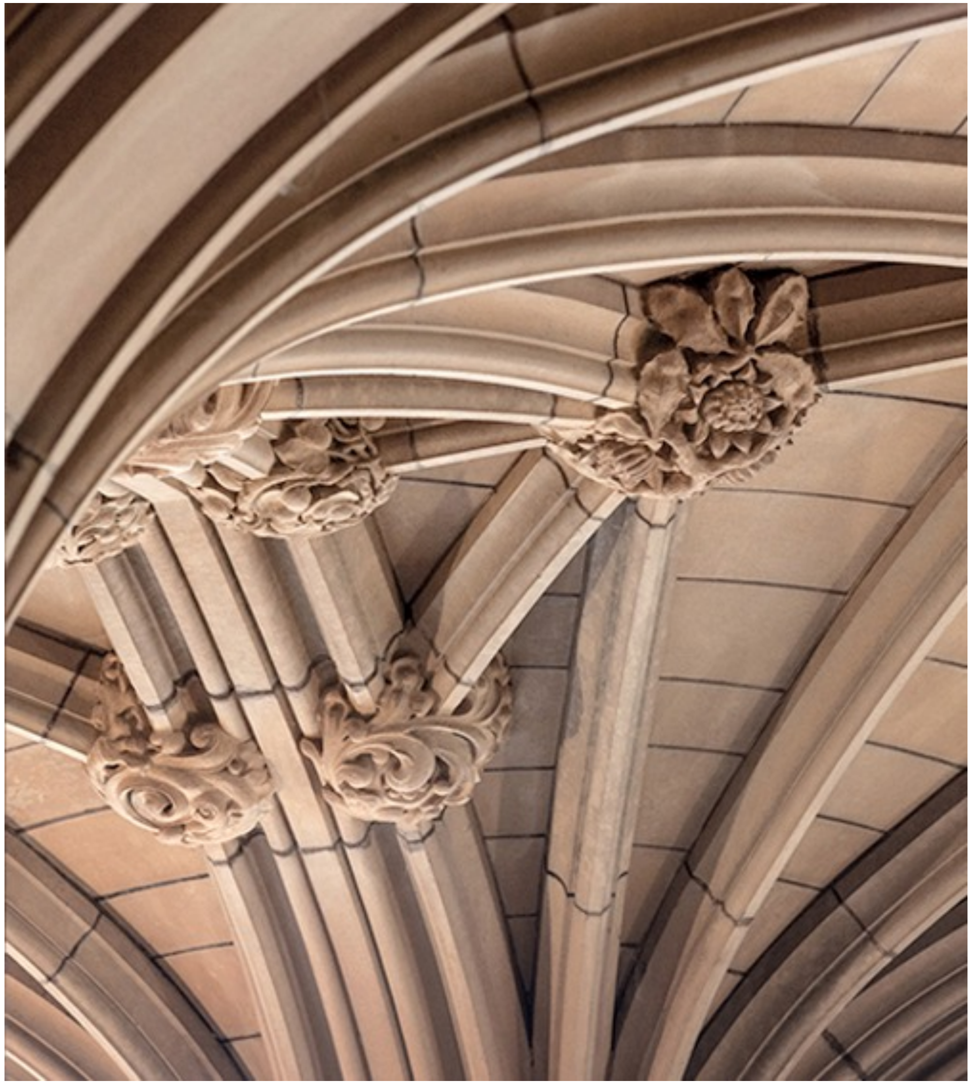
- Testing Types
  - Integration Testing, Regression Testing

- Advanced Testing Techniques
  - Black-box and White-box Testing
  - Test doubles (Dummies, Fakes, Stubs, Spies, Mocks)
  - Contract Test

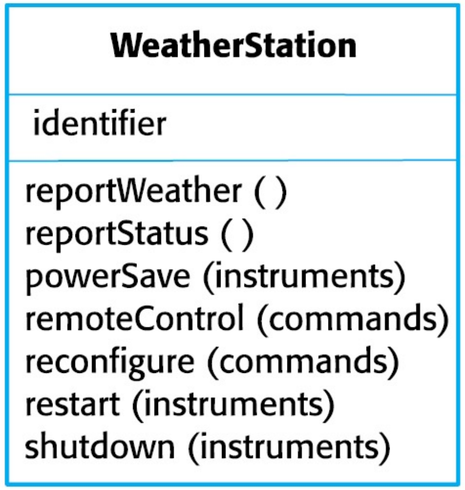
- Testing Frameworks
  - Mockito

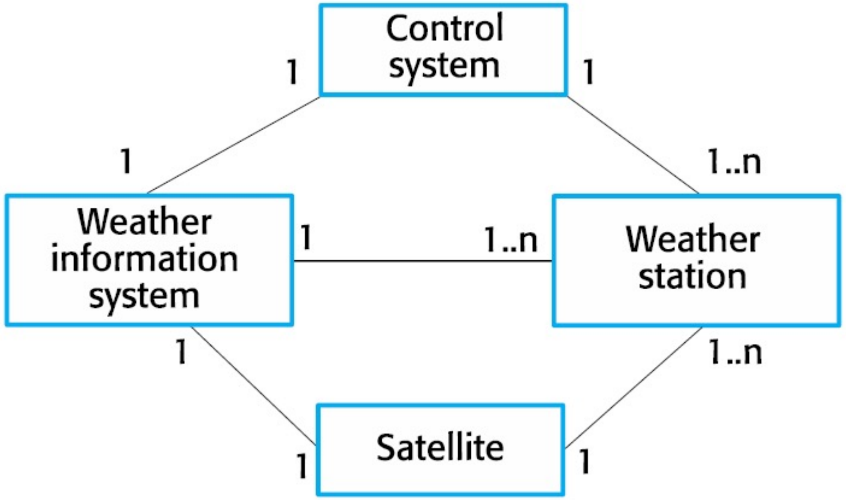
Advanced Testing Types

Integration testing, regression testing



Software Components/Sub-systems





- Potentially $O(n^2)$ interactions!
- Potentially $O(2^n)$ sub-systems!
- How to identify a sub-system to isolate interaction for testing?

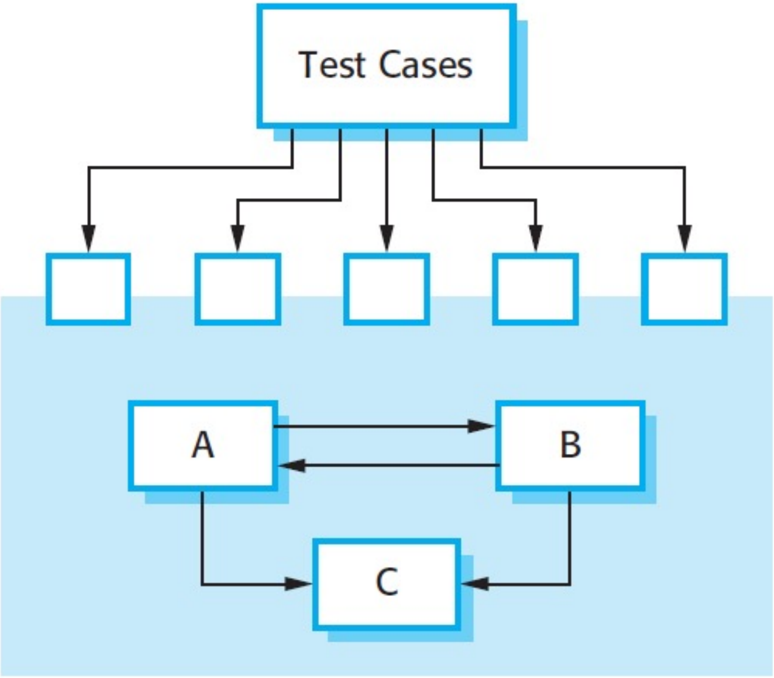
Integration Testing

- The process of verifying interactions/communications among software components behave according to its specifications
  - Problem large test-space / combinatorial explosion!
  - Dependencies are your friend!
  - Independently developed (and tested) units may not behave correctly when they interact with each other
- Incremental integration testing vs. “Big Bang” testing
  - **Incremental**: integrate components one by one using **stubs** or **drivers**
  - **Big Bang:** all the components are integrated in one shot

.

Integration Testing

- A, B and C integrated to create a software component/module
- Test cases should be designed to test the behavior of this module through its interface
- Testing may detect incorrect behavior that may result from interactions
- Example:
  - (A,B) – C as a stub
  - (A,C) – B as a driver/stub
  - (B,C) – A as a driver/stub



Types of interfaces

- Parameter interfaces
  - data/functions are passed from one component to another; e.g., methods
- Procedural interfaces
  - Encapsulates a set of procedures that can be called; e.g., interfaces in Java
- Message passing interfaces
  - service requests from a component to another component; e.g., OO world & client/server systems
- Shared memory interfaces
  - block of memory shared between components; R/W situations between different components; e.g., embedded system.

Interface Errors

- Interface misuse
  - error in use of interface
  - common for parameter interfaces
    - wrong type, order, number of arguments
- Interface misunderstanding
  - Calling component misinterprets specification of interface
  - Wrong assumptions about execution behavior
- Timing Errors
  - Real-time systems using shared-memory / message passing interfaces
  - Handling Out-of-time information

Incremental Testing

- Interaction between units are tested ***incrementally***
- Missing components replaced by
  - **Stubs:** modules that act as the lower-level modules that are not integrated yet
  - **Driver:** modules that act as the upper-level modules that are not integrated yet
- Advantage:
  - defects are found early in a smaller assembly
  - more thorough testing of the whole system
- Disadvantage:
  - Costs of testing is high / more effort is required.

Incremental Testing Methodologies

- **Top-down integration:**
  - from top to bottom w.r.t. dependencies
  - unavailable components/systems substituted by **stubs**
- **Bottom-up integration:**
  - from bottom to top w.r.t. dependencies
  - unavailable components/systems substituted by **drivers**
- **Functional incremental:**
  - merging units for testing individual functional requirements as per SW specification

Big-Bang Testing

- Units are tested in a complete system.
- Disadvantage:
  - difficult to isolate errors
    - no checks for verifying the interfaces across individual units
  - high probability of missing some critical defects
  - difficult to cover all the cases for integration testing
- Advantages:
  - No planning required
  - Suitable for small systems

Your Testing Exposed Bugs

- What would you do when your testing reveal bugs/errors?
- What would you do when you SW extensions 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 a software behaviour has not changed by incremental changes to the software
  - SW changes: for defect fixing or for enhancement.

- 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 still working as expected
  - No new bugs are introduced

- Types of Regression Test
  - **Final Regression Tests:** validation of the build for deployment/shipment
  - **Regression Tests:** the build hasn't broken any other parts by code changes

.

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

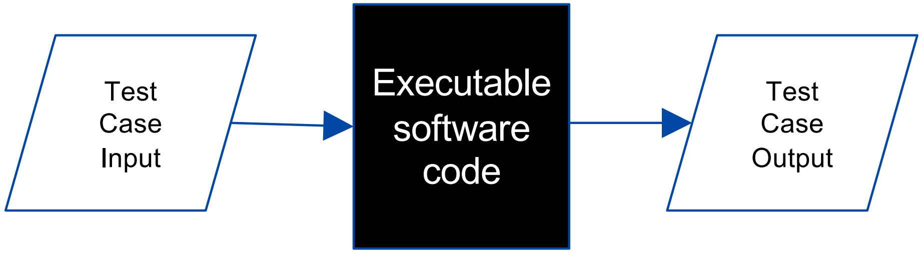
. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.460.5875&rep=rep1&type=pdf

Software Testing Techniques

Principle Testing Techniques

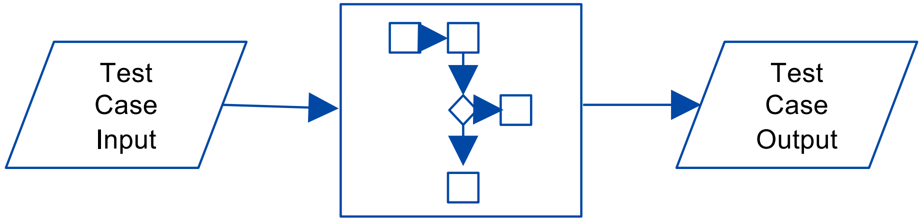
Black-box Testing

- no programming and software knowledge
- carried by software testers
- specifications based testing
- used for acceptance and system testing



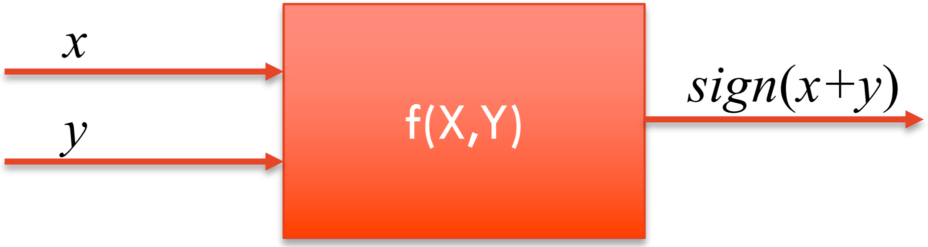
White-box Testing

- examines the program structure
- test developer must reason about implementation
- reveals “hidden” errors in the code



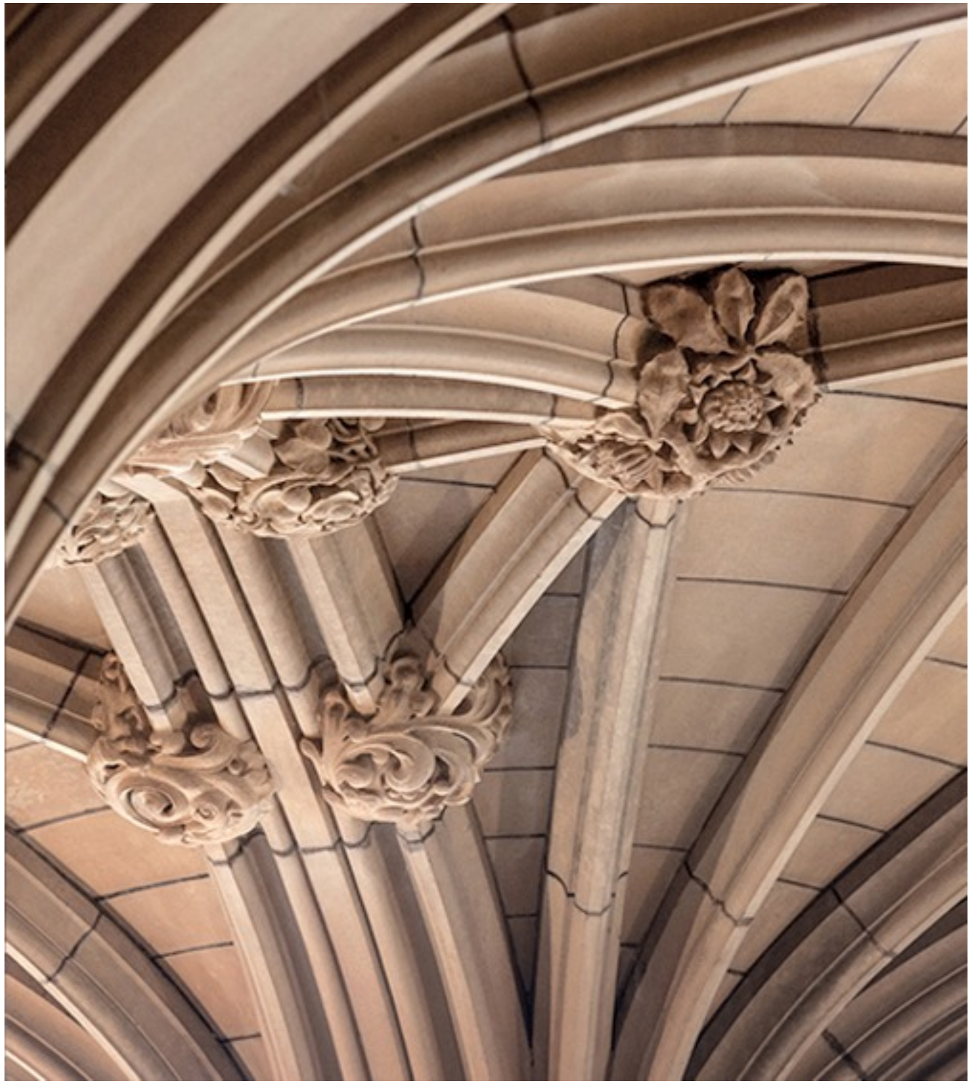
Black Box Testing – Example

- Test planned without knowledge of the code
- Based only on specification or design
- E.g., given a function that computes $sign\ (x+y)$



Advanced Testing Techniques/Methods

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

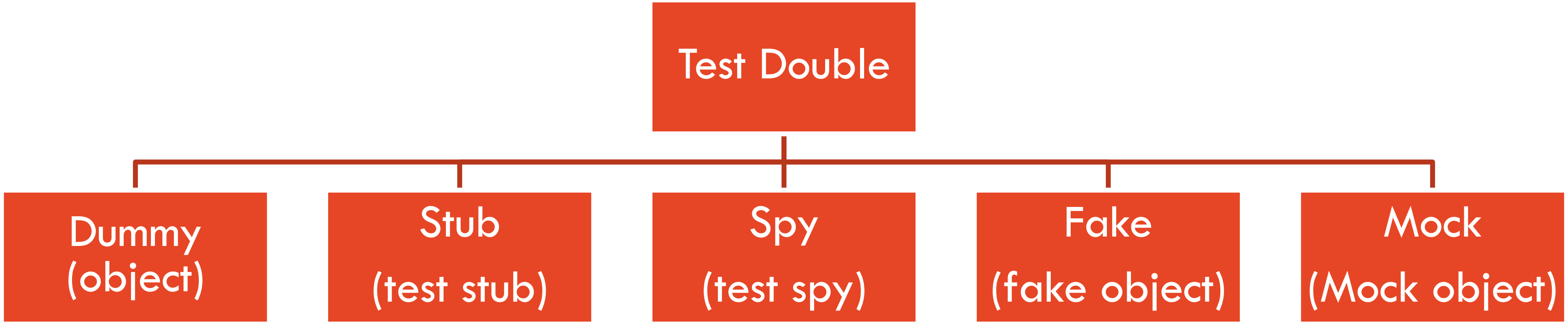
Test Double

- "*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

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

Test Double – Dummy Object



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 System Under Test (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

- Objects passed around but never actually used

- Usually used to fill parameter lists

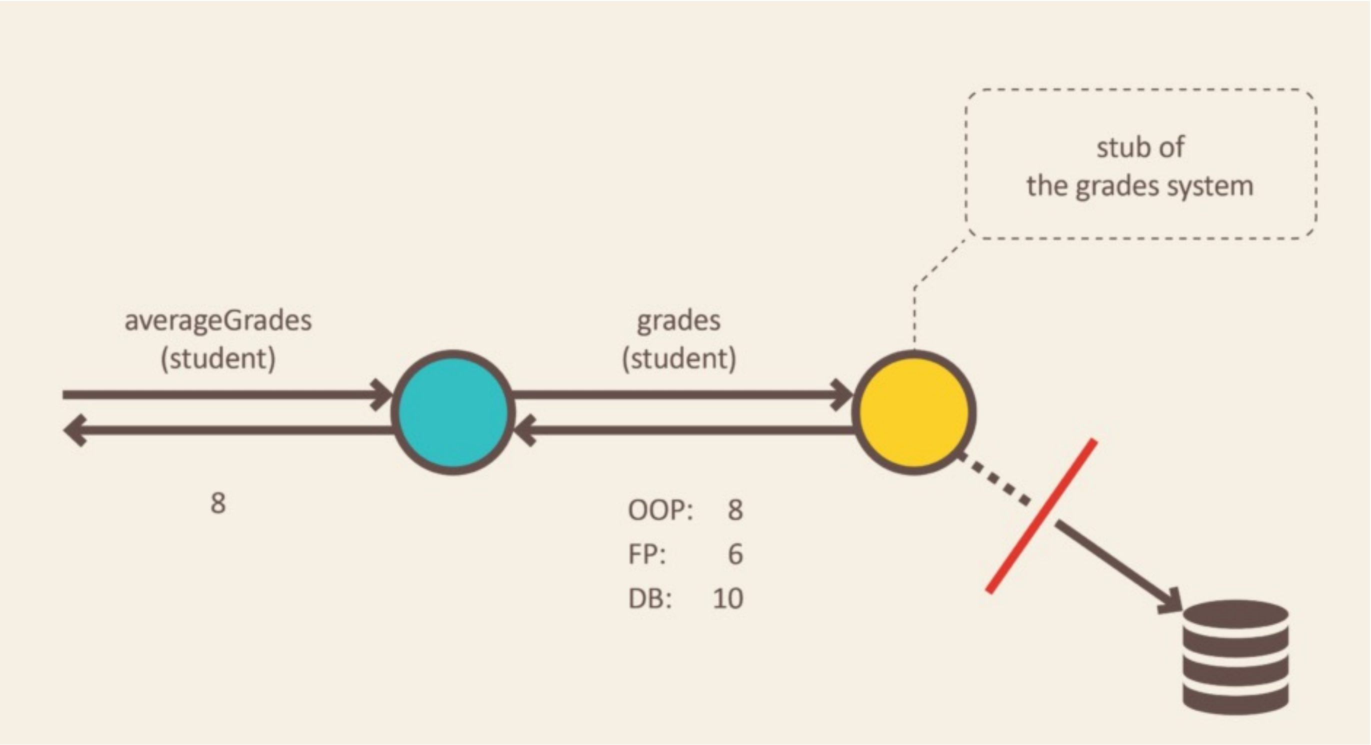
- Pass object with no implementation (dummy)
  - E.g., Fill in parameter lists

- 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

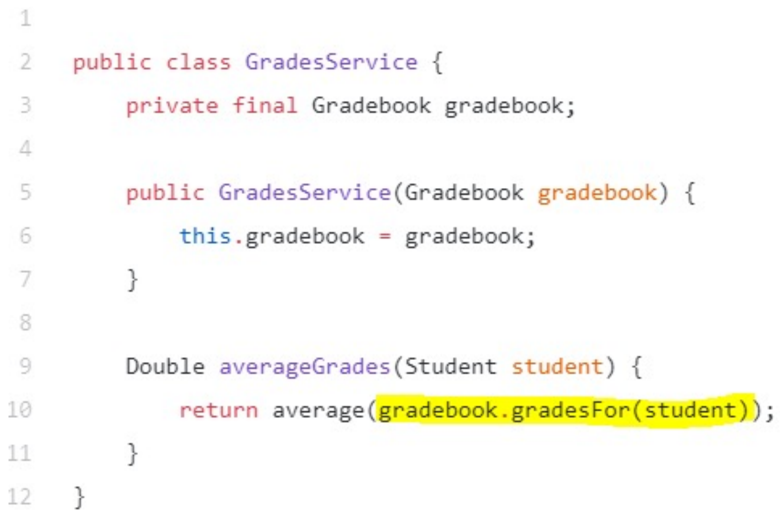
(Test) Stub

- Act as a lower-level module that is not yet integrated
- Provide canned answers to calls made during the test
- Responding to test workloads only
- A test-specific object that provides indirect inputs during tests
  - E.g., Object requires data from a database to answer a method call
- Control indirect inputs of the SUT using test stub

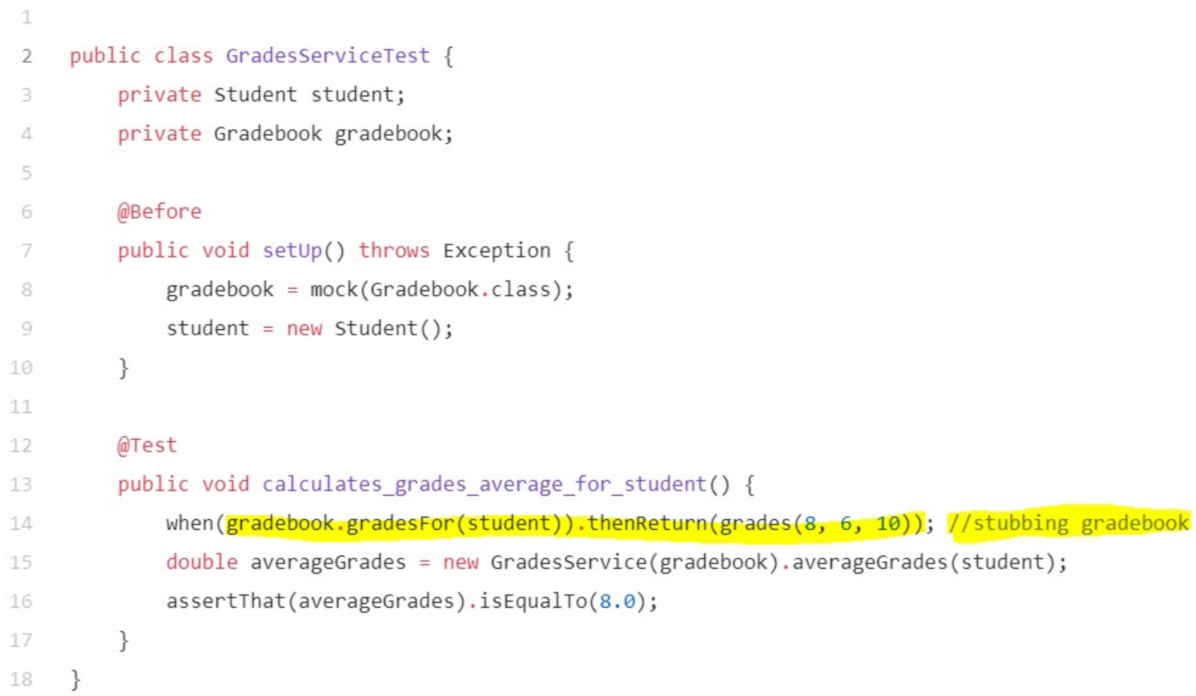
(Test) Stub – Example



(Test) Stub – Example Implementation



```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 = new Student();
    }

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

(Test) Spy

- Specific stubs recording information based on how they were called.

- Example: email service that records how many messages it was sent

- Capture output calls made by the SUT to another component for later verification

- Get enough visibility of the outputs generated by the SUT (observation point)

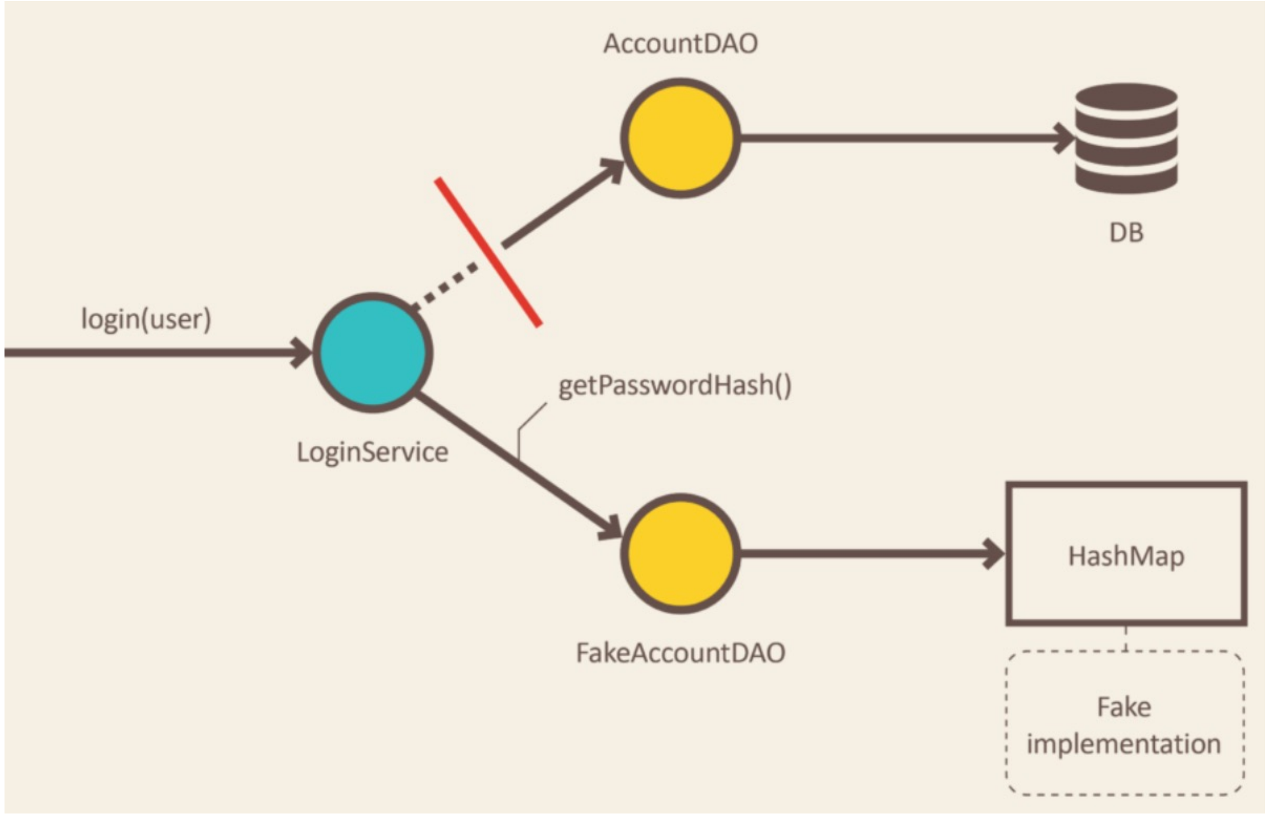
Fake (Object)

- Objects with working implementations but take shortcuts
- Example: InMemoryTestDatabase
- Objects to provide simplified implementation of a heavy (real) component
  - E.g., in-memory implementation of repository using simple collection to store data

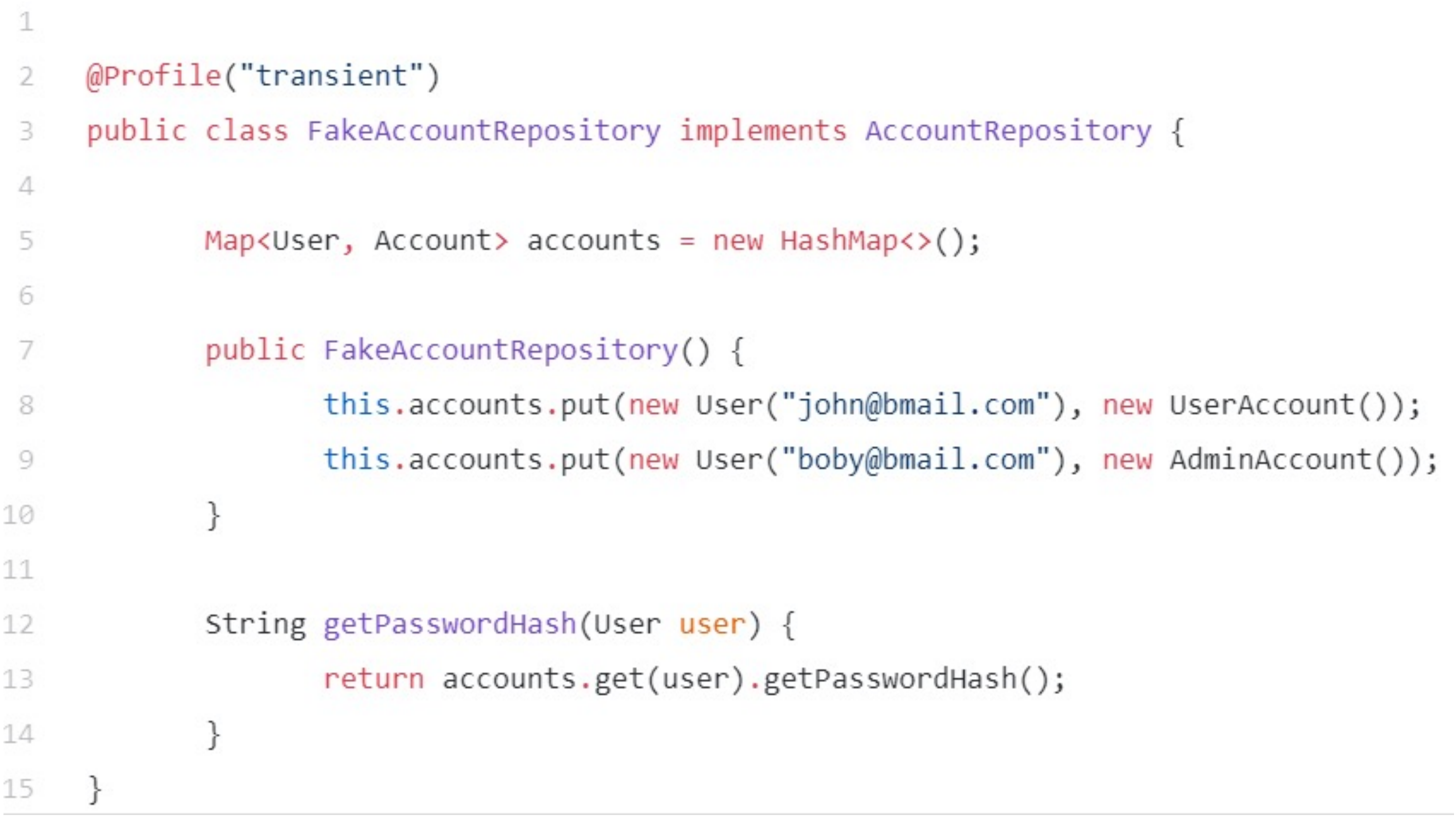
- SUT depends on other components that are unavailable or make testing complex or slow

- Should not be used when want to control inputs to SUT or outputs of SUT

Fake (Object) – Example



Fake (Object) – Example Implementation

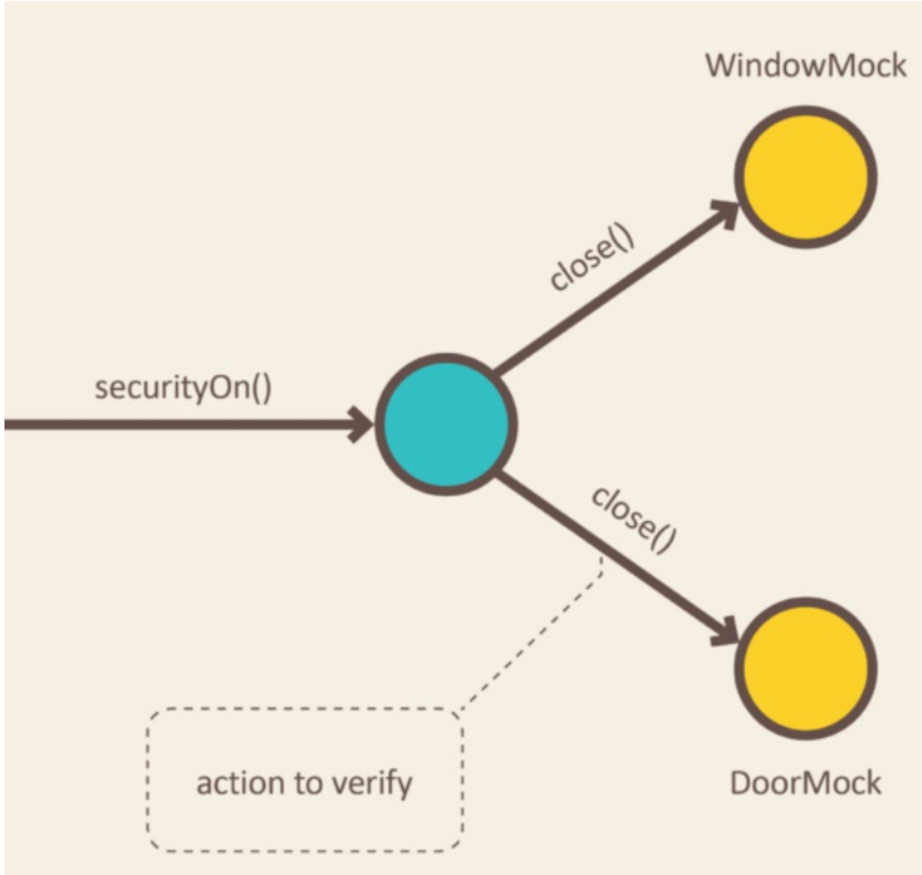


```java

@Profile("transient")
public class FakeAccountRepository implements AccountRepository {

        Map<User, Account> accounts = new HashMap<>();

        public FakeAccountRepository() {
                this.accounts.put(new User("john@bmail.com"), new UserAccount());
                this.accounts.put(new User("boby@bmail.com"), new AdminAccount());
        }

        String getPasswordHash(User user) {
                return accounts.get(user).getPasswordHash();
        }
}
```

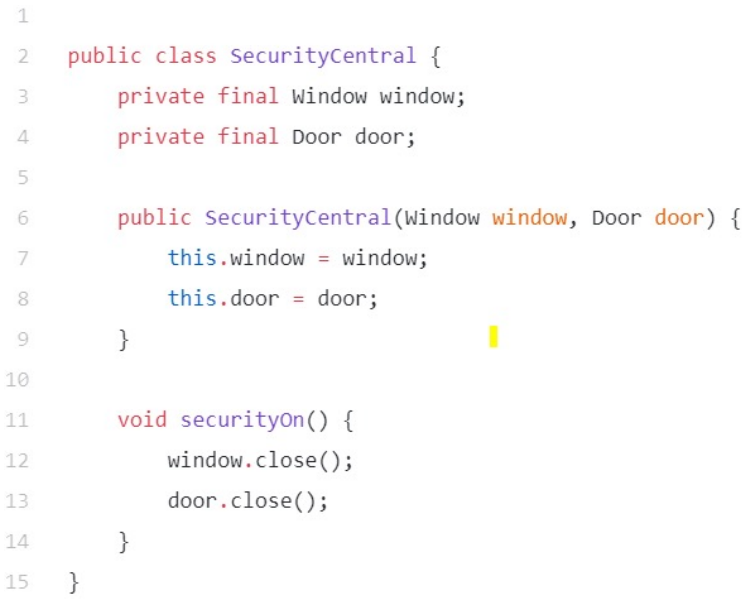
Mock (Object)

- Pre-programmed with expectations form a specification of the calls they are expected to receive
- Throw an exception if they receive a call they don't expect and are checked during verification to ensure they got all the calls they were expecting.
- Object(s) that verify indirect output of the tested code
  - E.g., function that calls email sending service, not to really send emails but verify that email sending service was called
- Calling real implementation during testing is tedious, or the side effect is not the testing goal

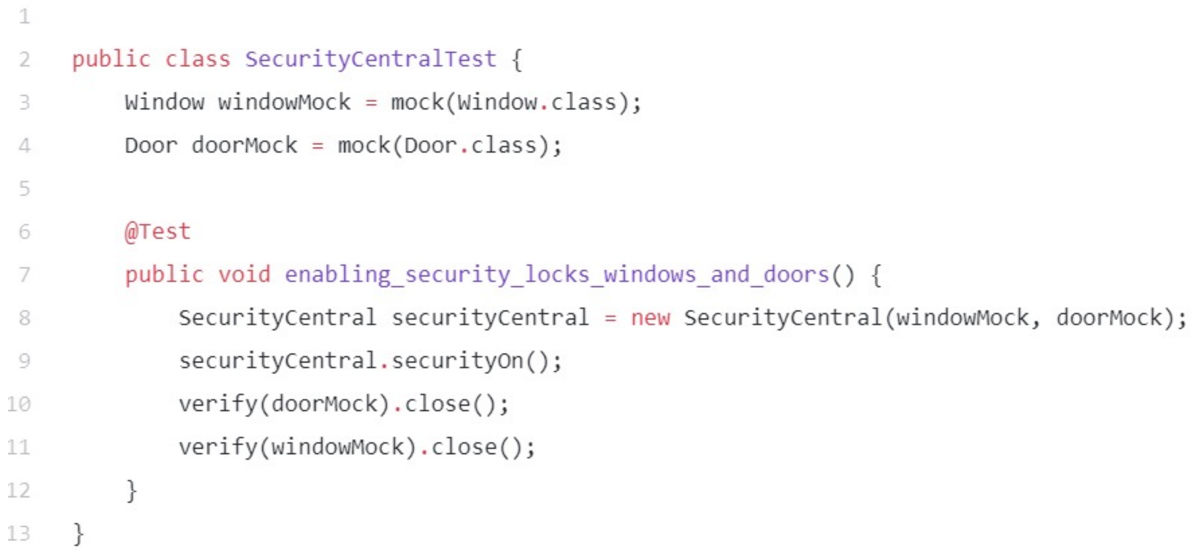
Mock (Object) – Example



Mock (Object)

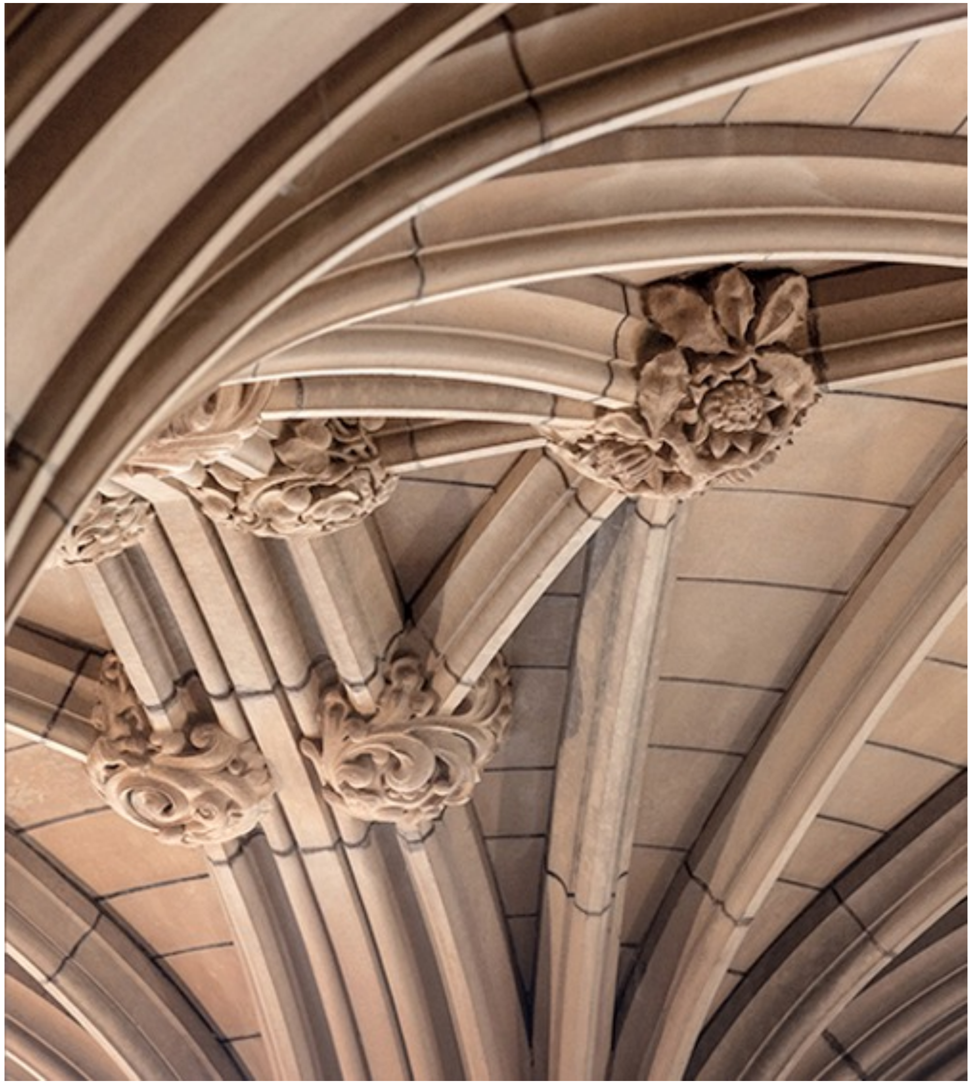


```java

public class SecurityCentral {
    private final Window window;
    private final Door door;

    public SecurityCentral(Window window, Door door) {
        this.window = window;
        this.door = door;
    }

    void securityOn() {
        window.close();
        door.close();
    }
}
```



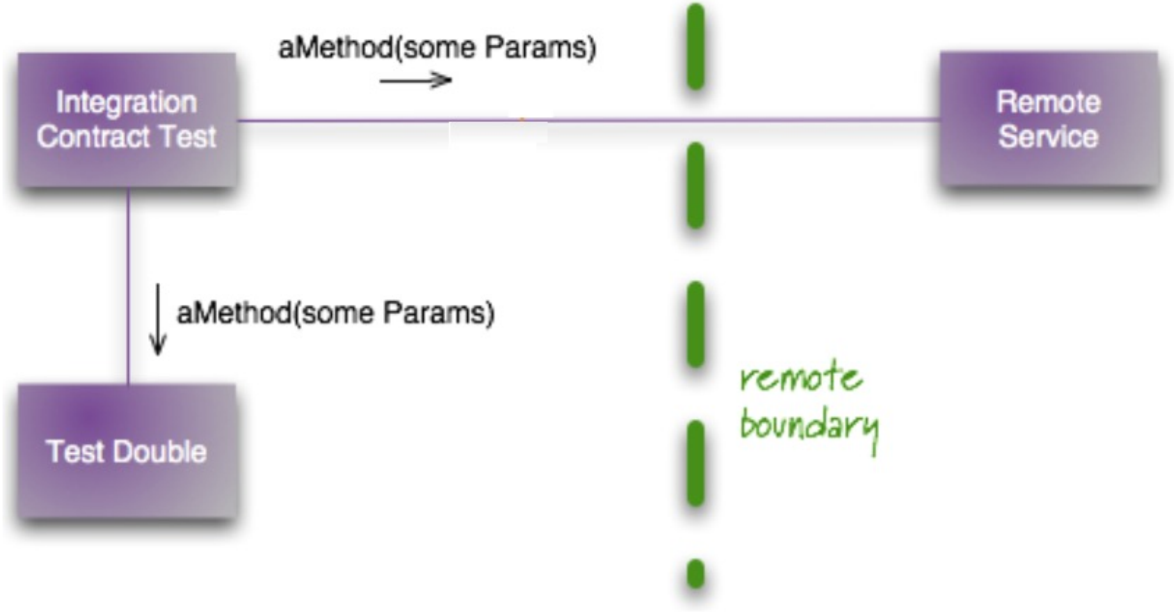
```java

public class SecurityCentralTest {
    Window windowMock = mock(Window.class);
    Door doorMock = mock(Door.class);

    @Test
    public void enabling_security_locks_windows_and_doors() {
        SecurityCentral securityCentral = new SecurityCentral(windowMock, doorMock);
        securityCentral.securityOn();
        verify(doorMock).close();
        verify(windowMock).close();
    }
}
```

Contract Test



Test Double – External Services

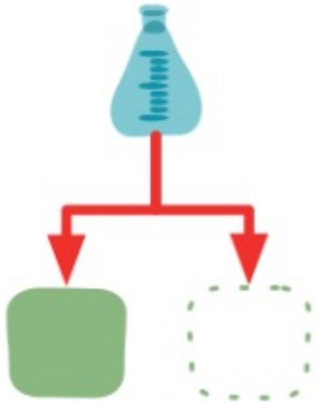
- Test double to interact with external/remote service
  - How accurate/reliable is a test double?



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

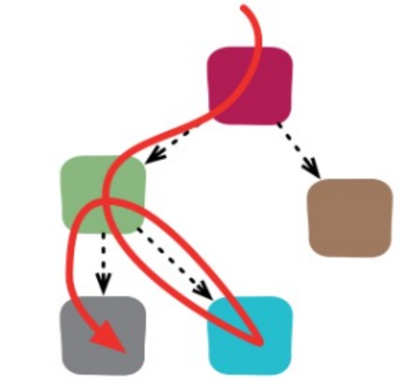
Contract Test

- 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

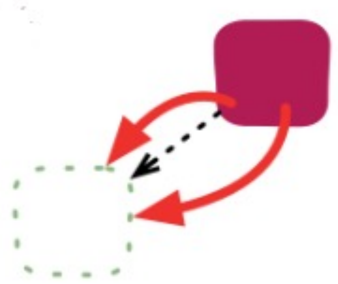


https://martinfowler.com/bliki/ContractTest.html

Integration Testing – Broad vs. Narrow Tests

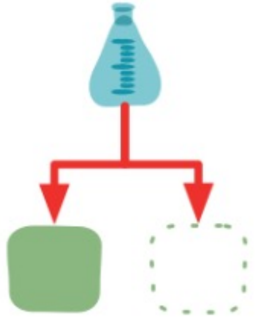


“Broad tests done with many modules active”



“Narrow tests of interaction with individual test doubles”

+

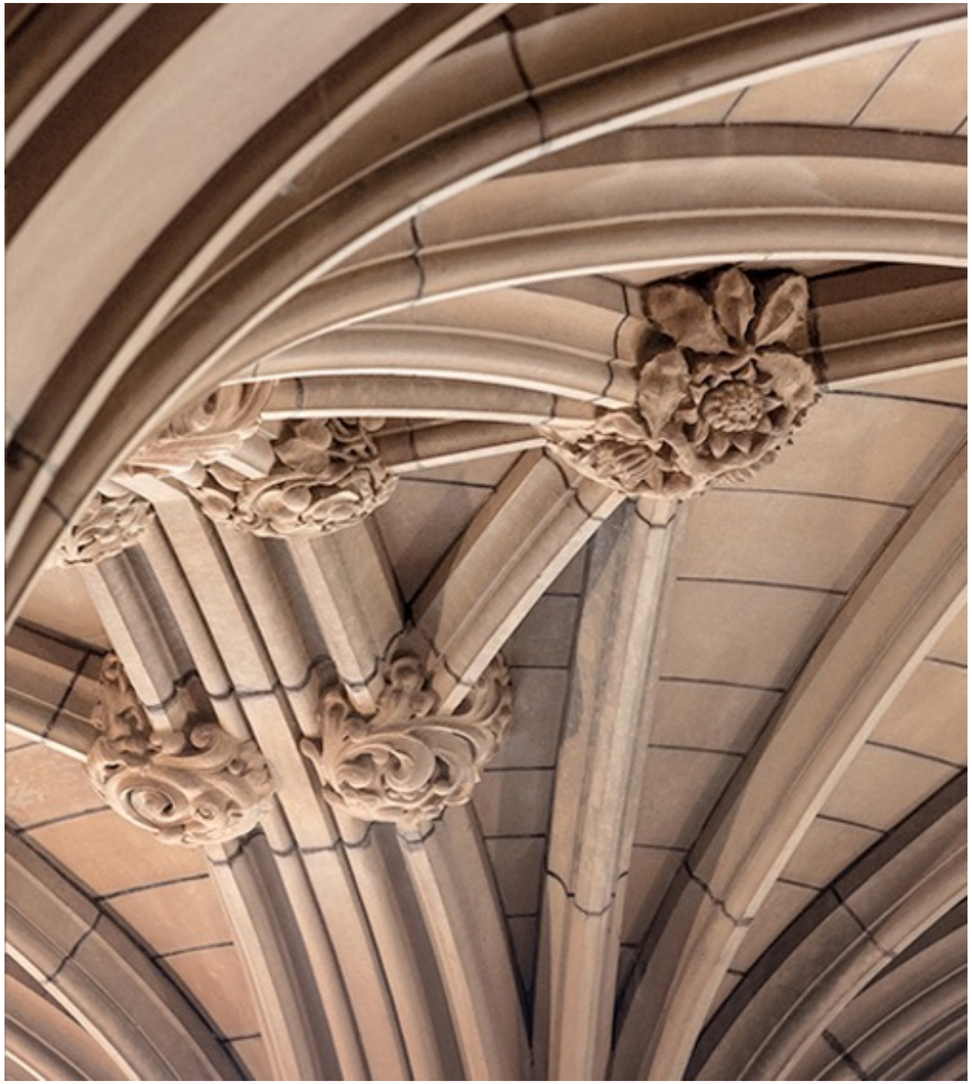


“supported by *contract tests* to ensure the faithfulness of the double”

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

Testing Frameworks/ Tools

Mockito



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 state (unit) testing
- Allows to specify order of verification (not all interactions)

LINK

[1] 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	Creates a spy 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

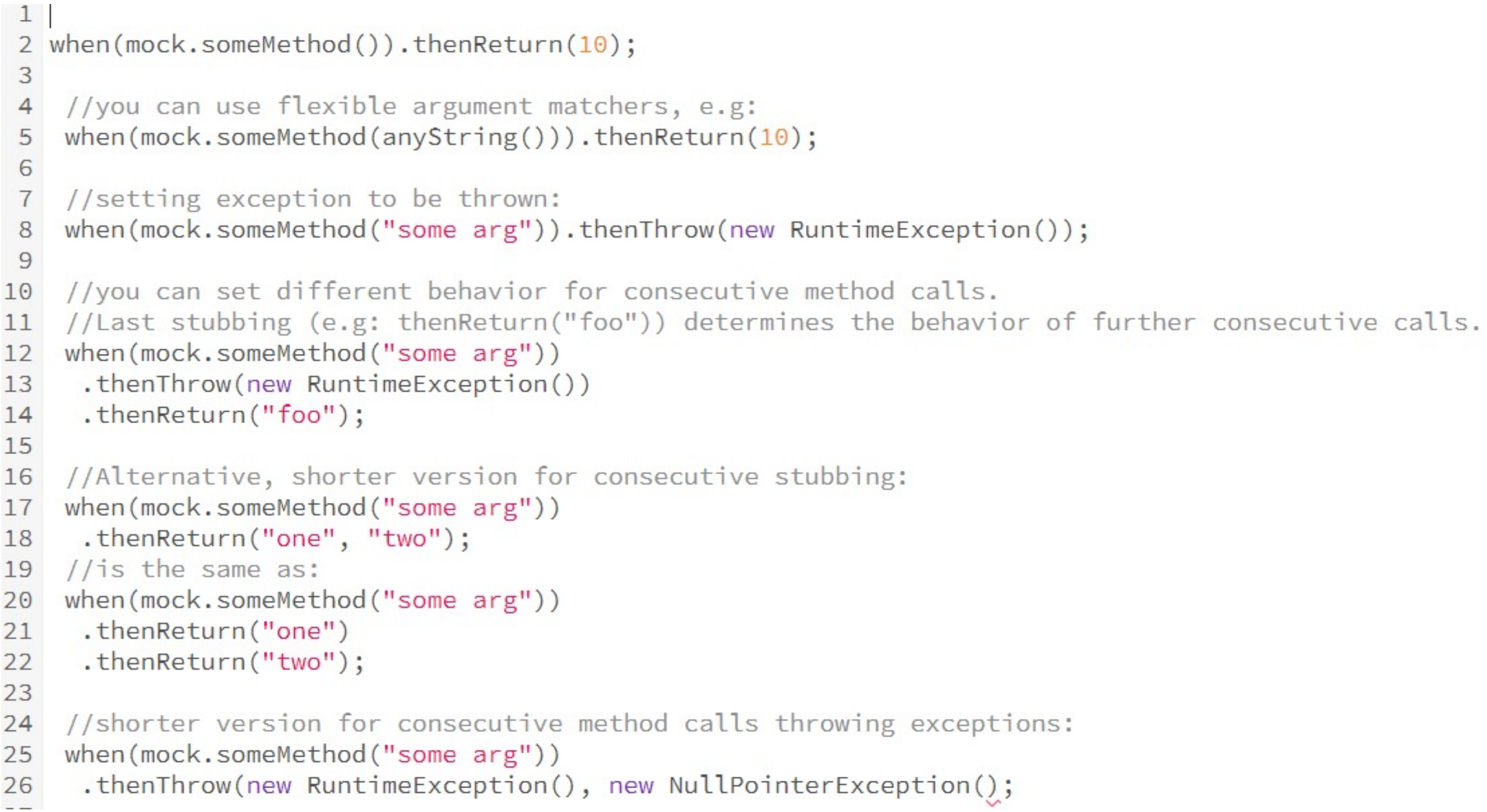
http://static.javadoc.io/org.mockito/mockito-core/2.24.0/org/mockito/Mockito.html

Mockito – 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



```java

when(mock.someMethod()).thenReturn(10);

 //you can use flexible argument matchers, e.g:
 when(mock.someMethod(anyString())).thenReturn(10);

 //setting exception to be thrown:
 when(mock.someMethod("some arg")).thenThrow(new RuntimeException());

 //you can set different behavior for consecutive method calls.
 //Last stubbing (e.g: thenReturn("foo")) determines the behavior of further consecutive calls.
 when(mock.someMethod("some arg"))
  .thenThrow(new RuntimeException())
  .thenReturn("foo");

 //Alternative, shorter version for consecutive stubbing:
 when(mock.someMethod("some arg"))
  .thenReturn("one", "two");
 //is the same as:
 when(mock.someMethod("some arg"))
  .thenReturn("one")
  .thenReturn("two");

 //shorter version for consecutive method calls throwing exceptions:
 when(mock.someMethod("some arg"))
  .thenThrow(new RuntimeException(), new NullPointerException();
```

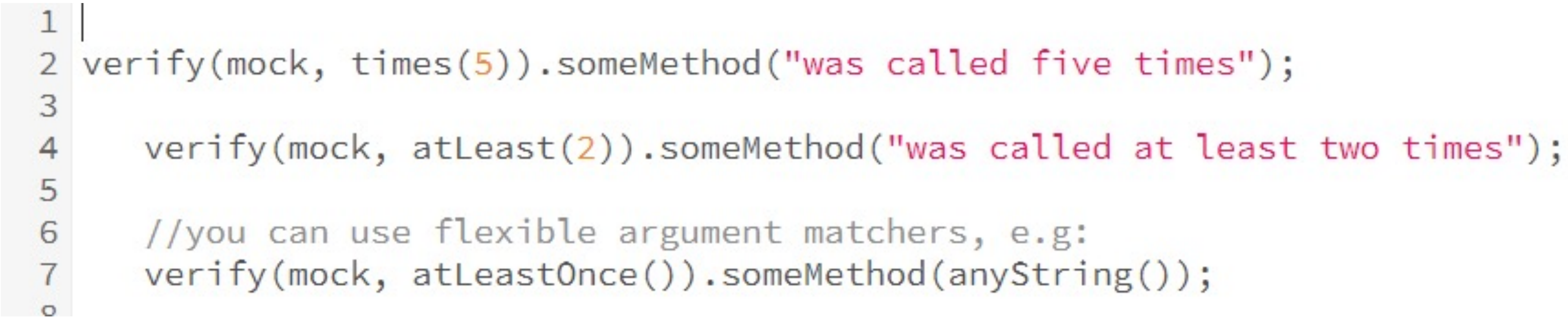
http://static.javadoc.io/org.mockito/mockito-core/2.24.0/org/mockito/Mockito.html#when-T-

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

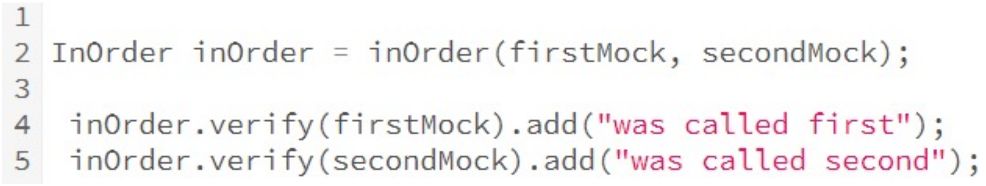


```

verify(mock, times(5)).someMethod("was called five times");

   verify(mock, atLeast(2)).someMethod("was called at least two times");

   //you can use flexible argument matchers, e.g:
   verify(mock, atLeastOnce()).someMethod(anyString());
```

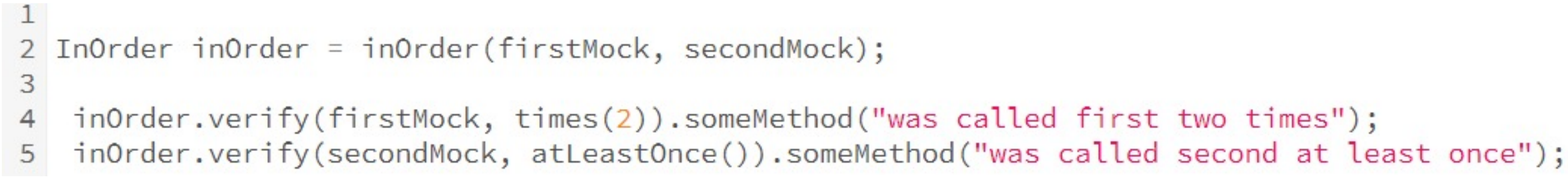
- Default mode is times (1) which can be omitted
- Argument passed are compared suing equals() method

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



```java

InOrder inOrder = inOrder(firstMock, secondMock);

 inOrder.verify(firstMock).add("was called first");
 inOrder.verify(secondMock).add("was called second");
```



```java

InOrder inOrder = inOrder(firstMock, secondMock);

 inOrder.verify(firstMock, times(2)).someMethod("was called first two times");
 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

Writing Good Tests

- Readable
  - Follow recommended coding practices (e.g., naming conventions, documentation)
- Reliable
  - Free of bugs/defects

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

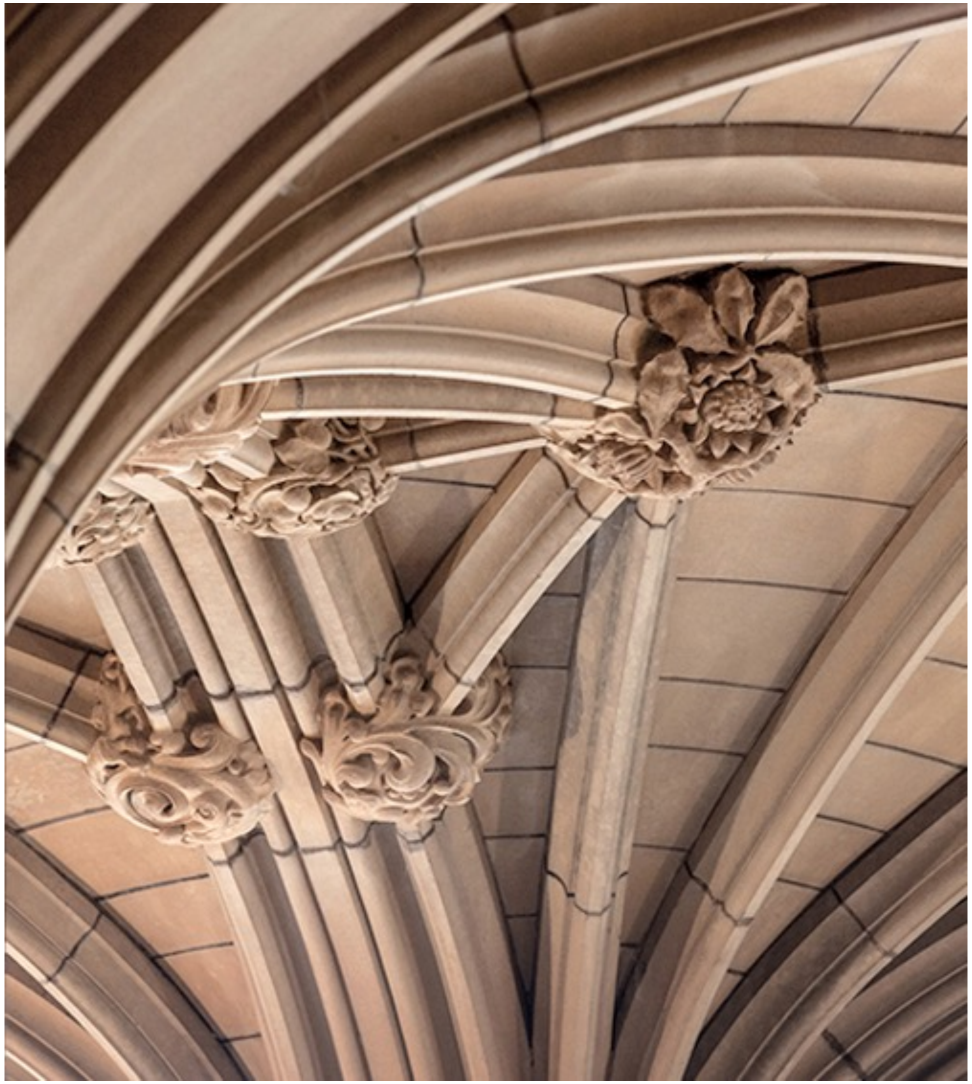
References

- Ian Sommerville. 2016. Software Engineering (10th ed.) Global Edition. Pearson, Essex England

- Martin Fowler, various testing articles. https://martinfowler.com/

- Michal Lipski, Pragmatists: Test doubles: Fakes, Mocks and Stubs. https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da

Software Design/Modelling & Construction



THE UNIVERSITY OF SYDNEY