JOHNS HOPKINS
UNIVERSITY

EN.601.422 / EN.601.622

Software Testing & Debugging

Double

Think “stunt double”

Test Double

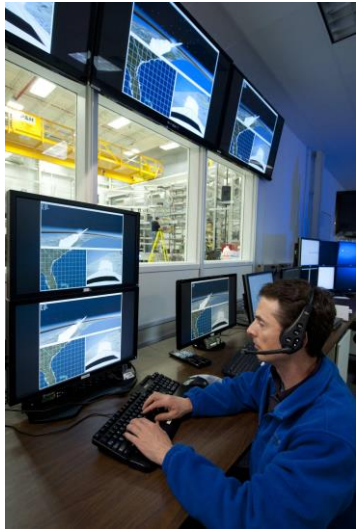
- Test Double is a software component (method, class, collection of classes etc.) that implement partial functionality.
- Test Double are used if some component:
  - is not available or implemented yet
  - will result in unrecoverable actions (BOMB effect!)
  - is expensive to run (e.g., takes considerable amount of time)
  - is too difficult to instantiate or configure

Test Double

- Test doubles must be implemented with as little change as possible to the software
- Terminology:
  - **Stubs**: a skeletal or special purpose implementation that typically provides canned answers to calls made during the test
  - **Mocks**: A pre-programmed class with ability to verify if the class under test made correct interactions with them
  - **Fakes**: objects with working implementations which “take shortcut” when needed
  - **Dummys**: objects that are passed around but never actually used

Examples – Bomb effect

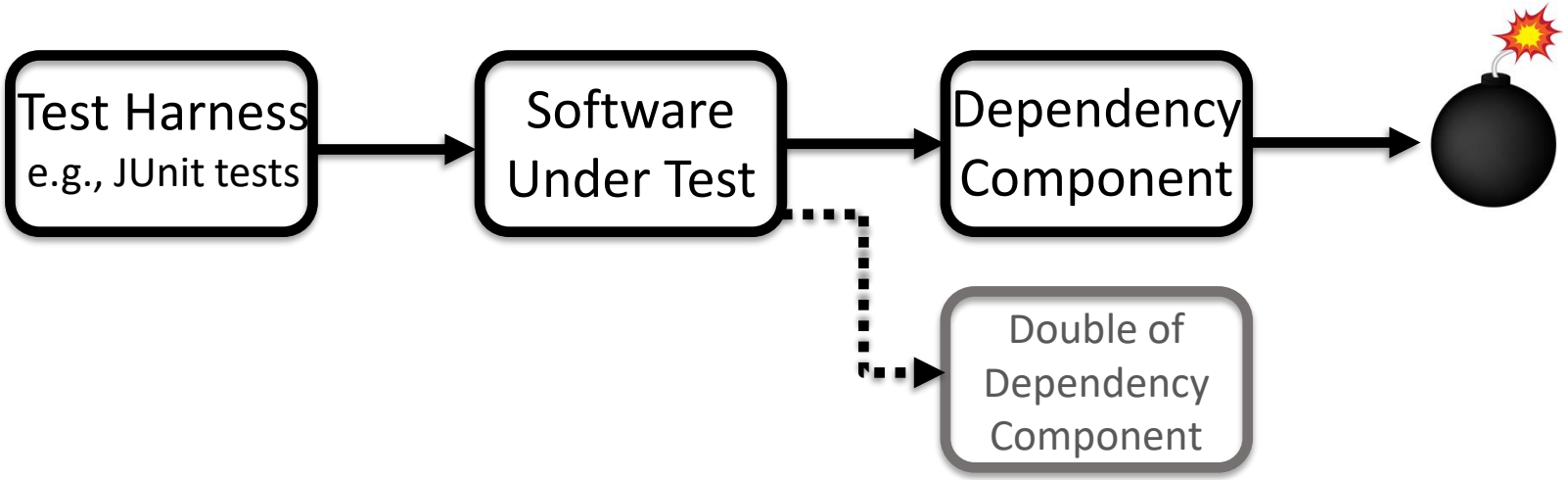
- Software systems that:
  - send emails to lots of customers
  - performs financial transactions (e.g., charge a credit card)
  - launches a missile!
  - etc.



More Examples

- **Test-Driven Development**: tests are written and executed prior to the implementation
- Database is not ready yet or a live db connection does not exist at the time of testing
- Heavy calculations that take considerable amount of time to finish
- Need user interaction

Testing with Doubles



Testing with Doubles

- Typical workflow:
  1. obtain/create necessary test doubles (using mocking tools)
  2. specify the expected sequence of interactions with the test double
  3. carry out the action under test
  4. verify that the expected interactions, in fact, occurred

Testing with Doubles

- Many tools exist to help with this:
  - Most IDEs automatically generate simple stubs
  - Automated Mock testing tools:
    - **Java**: Mockito, jMock, EasyMock, PowerMock etc.
    - **C++**: Google Mock, TypeMock
    - **Python**: unittest.mock, PyMock, Mox, etc.
    - **etc.**

Class Order

```java
public class Order {
    String product;
    int count;
    boolean isFilled;

    public Order(String product, int count) {
        super();
        this.product = product;
        this.count = count;
        isFilled = false;
    }
    public void fill(Warehouse wh) {
        // implementation of fill
        // set isFilled on success
        if (wh.hasInventory(product, count)) {
            wh.remove(product, count);
            isFilled = true;
        } else {
            isFilled = false;
        }
    }
    public boolean isFilled() {
        return isFilled;
    }
}
```

Interface Warehouse

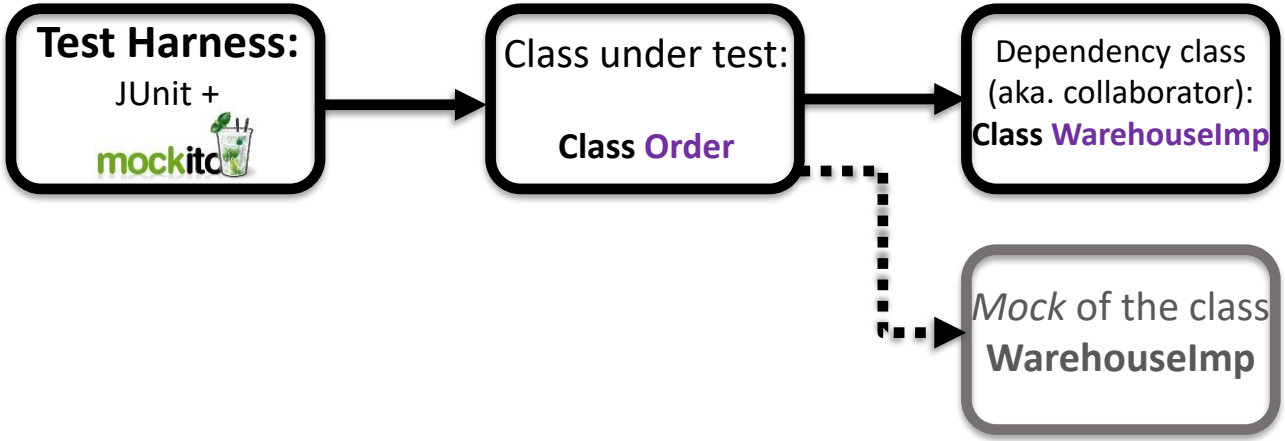
```java
public interface Warehouse {
    // Warehouse interface
    int getInventory(String product);
    boolean hasInventory(String product, int count);
    void add(String product, int i);
    void remove(String product, int count);
}
```

Class Warehouse

```java
public class WarehouseImpl implements Warehouse {
    public int getInventory(String product) {
        // getInventory implementation
        return -1; // STUB
    }
    public boolean hasInventory(String product, int count) {
        // hasInventory implementation
        return false; // STUB
    }
    public void remove(String product, int count) {
        // remove implementation
    }
    public void add(String product, int count) {
        // add implementation
    }
}
```

Example

- Assume **"Warehouse"** is not implemented yet, is not available, is very costly to instantiate from for testing, or making queries takes considerable time, etc. ➔ **Use a double (*mock* in this case)**



State-based JUnit Tests

```
public class OrderTest {
  private static String TALISKER = "Talisker";
  private static String HIGHLAND_PARK = "Highland Park";
  private Warehouse warehouse;
  @BeforeEach
  public void setUp() throws Exception {
    warehouse = new WarehouseImpl();
    warehouse.add(TALISKER, 50);
    warehouse.add(HIGHLAND_PARK, 25);
  }
  @Test
  public void testOrderIsFilledIfEnoughInWarehouse() {
    Order order = new Order(TALISKER, 50);
    order.fill(warehouse);
    assertTrue(order.isFilled());
    assertEquals(0, warehouse.getInventory(TALISKER));
  }
  @Test
  public void testOrderDoesNotRemoveIfNotEnough() {
    Order order = new Order(TALISKER, 51);
    order.fill(warehouse);
    assertFalse(order.isFilled());
    assertEquals(50, warehouse.getInventory(TALISKER));
  }
}
```

verifying the output (i.e., state)

Behavior-based “Mock” testing

```
public class OrderMockTest {
    private static String TALISKER = "Talisker";
    private static String HIGHLAND_PARK = "Highland Park";
    private Warehouse warehouseMock;
    @BeforeEach
    public void setUp() throws Exception {
        warehouseMock = mock(WarehouseImpl.class);
        warehouseMock.add(TALISKER, 50);
        warehouseMock.add(HIGHLAND_PARK, 25);
    }
    @Test
    public void testOrderFilledCallsInventoryAndRemove() {
        when(warehouseMock.hasInventory(TALISKER, 50)).thenReturn(true);
        Order order = new Order(TALISKER, 50);
        order.fill(warehouseMock);
        verify(warehouseMock, times(1)).hasInventory(TALISKER, 50);
        verify(warehouseMock, atLeast(1)).remove(TALISKER, 50);
    }
    @Test
    public void testOrderNotFilledCallsOnlyhasInventory() {
        when(warehouseMock.hasInventory(TALISKER, 51)).thenReturn(false);
        Order order = new Order(TALISKER, 51);
        order.fill(warehouseMock);
        verify(warehouseMock, times(1)).hasInventory(TALISKER, 51);
        verify(warehouseMock, never()).remove(anyString(), anyInt());
    }
}
```

verifying the **interaction** with “collaborator” (i.e., behavior)

Behavior-based “Mock” testing

```
public class OrderMockTest {
  private static String TALISKER = "Talisker";
  private static String HIGHLAND_PARK = "Highland Park";
  private Warehouse warehouseMock;
  @BeforeEach
  public void setUp() throws Exception {
    warehouseMock = mock(WarehouseImpl.class);
    warehouseMock.add(TALISKER, 50);  warehouseMock.add(HIGHLAND_PARK, 25);
  }
  @Test
  public void testOrderIsFilled {
    when(warehouseMock.hasInventory(TALISKER, 50)).thenReturn(true);
    Order order = new Order(TALISKER, 50);
    order.fill(warehouseMock);
    InOrder inOrder = inOrder(warehouseMock);
    inOrder.verify(warehouseMock, times(1)).hasInventory(TALISKER, 50);
    inOrder.verify(warehouseMock, times(1)).remove(TALISKER, 50);
  }
  @Test
  public void testOrderNotFilledCallsOnlyhasInventory() {
    when(warehouseMock.hasInventory(TALISKER, 51)).thenReturn(false);
    Order order = new Order(TALISKER, 51);
    order.fill(warehouseMock);
    InOrder inOrder = inOrder(warehouseMock);
    inOrder.verify(warehouseMock, times(1)).hasInventory(TALISKER, 51);
    inOrder.verify(warehouseMock, never()).remove(anyString(), anyInt());
  }
}
```

verifying the **order of interactions**

Mock vs. Stub

- Stub:
  - State-based verification
  - Provides canned answers
    - Example: return 0 from a method that should return int, null from a method that should return an object, etc.
- Mock:
  - Behavior-based verification (aka interaction-based verification)
  - Verifies certain interactions were made with the mock (in a certain order)

Behavior- vs. State-based Testing

- If the dependency is available and undemanding:
  - Use real class and state-based testing
- If the dependency is not available and/or demanding:
  - Use real class and state-based if possible
  - Otherwise, use interaction-based testing

Behavior- vs. State-based Testing

- Behavior-based testing tests the outbound calls of the SUT to ensure it talks properly to its collaborators:
  - More coupling to dependencies implementations since it checks all the interactions, the order of interactions etc.
  - If implementation of collaborators change (e.g., method names, method parameters etc.), behavior-based tests are more likely to break
- State-based, on the other hand, focuses on the final state:
  - Checks exact output values
  - Less coupling to dependencies implementations

Relevant Reads and Resources

- Recommended texts:
  - Introduction to Software Testing, 2nd Edition: ch 12
- Mocks aren't Stubs by Martin Fowler:
  - https://martinfowler.com/articles/mocksArentStubs.html
- https://site.mockito.org/

JOHNS HOPKINS

WHITING SCHOOL
of ENGINEERING