Double Objects

Mariana Bravo
marivb@agilcoop.org.br
Translation: Paulo Cheque

Summer 2009

Motivation

- Unit tests focus on an unit
- But units have dependencies

- Double objects offers a way to isolate dependencies

Dependencies

- **Indirect input** – Data obtained from a dependency object
(instance attribute, parameter, etc)

- **Indirect output** – Expected results can not be verified by the return value
(Colateral effects)

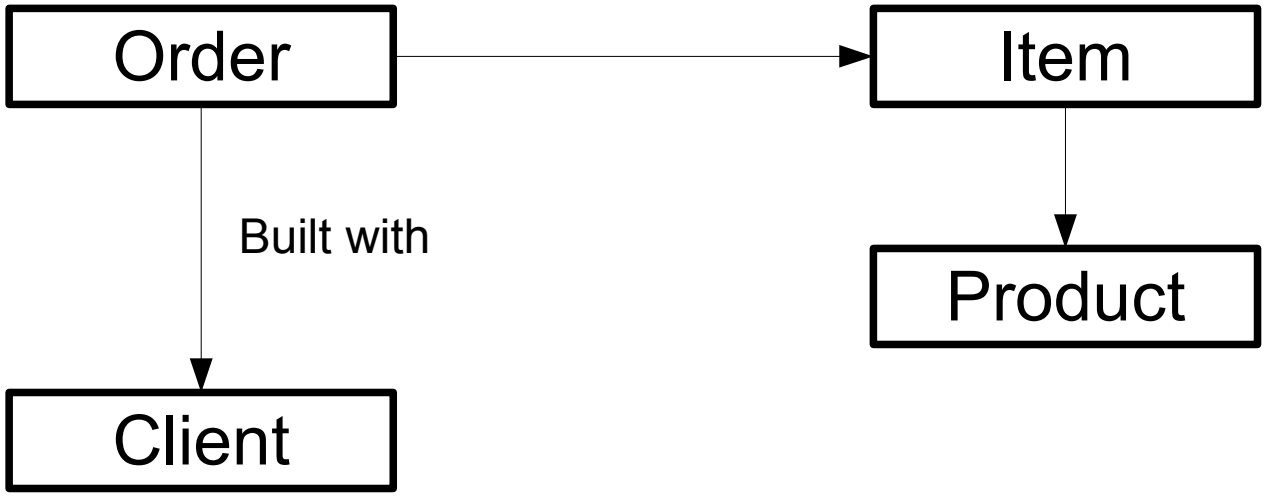
Definition

- A double object is a specialization of some component on which a unit test depends

Types of Doubles

- *Dummy object*: allows creation/execution of a test case
- *Test stub*: Provides information of a unit under test
- *Test spy*: Captures and stores indirect calls
- *Mock object*: verifies indirect calls and provides data
- *Fake object*: contains a special and fake implementation of a real component

Example – Dummy Object



- In Order test, I want to verify if addition of products works:
  - `order.addAmount(produto, 1);`
    `assertEquals(1, order.getItems().size());`
    `assertEquals(expectedItem, items.get(0));`
- To create an Order I need a Client, but a Client is not actually needed to add Products

Dummy Object

Allows the creation of a test to satisfy some call

But it is not used at any moment by the unit under test

Example – Test Stub

- Class TimeFormatter provides current hour in different formats

```
public String currentTimeInWriting() {
  Calendar now = Calendar.getInstance();
  if (now.get(Calendar.HOUR_OF_DAY) == 0)
    return "Midnight";
  else if (now.get(Calendar.HOUR_OF_DAY) == 12)
    return "Noon";
  else
    return now.get(Calendar.HOUR_OF_DAY) + " hours";
}
```

- How to test it?

Test Stub

Allows to control indirect input of unit under test, **providing data** that could be hard to get from a real component

Example – Test Stub Saboteur

- Sometimes we want to test exceptional flows of a program

```
public void readInformation(Reader reader)
{
  try {
    while (reader.read() != 0) {
      // Process the information
      // e store something
    }
  } catch (IOException e) {
    // Handle exception
  }
}
```

- How to test if that catch is correct?

Example – Test Spy

- Continuing previous example:

```
catch (IOException e) {
  // Handle exception
  logger.log("File read error", e);
}
```

- How to test if an error is logged correctly?

Test Spy

Captures information about collateral effects caused by unit under test to verify if they are correct

Another Example – Test Spy

- Class Shape that must notify observers whenever it is modified

```
public void addPicture(Picture picture) {
   pictures.add(picture);
   setChanged();
   notifyObservers(picture);
}
```

- How to test this situation?

Example – Mock Object

- Just like the previous example! But with another approach to test.

Mock Object

Verify directly collateral effects caused by unit under test

Test Spy vs. Mock Object

- Test Spy: verification of behavior
  - Unit under test is called
  - Spy captures information
  - Test verifies if it is correct
- Mock Object: specification of expected behavior
  - Mock is loaded with expected calls
  - Call unit under test with mock object

Types of Mock Object

- Strict:
  It waits for calls in an exactly specified order that was specified

- Tolerant:
  Accepts calls in any order, including variable number of calls

Example with EasyMock

- From the previous example

Example – Fake Object

- Java BufferedReader reads line by line. We want a special Reader that reads an entire paragraph (sequence of empty lines)

```
public class SpecialReader extends BufferedReader {

  public MySpecialReader(Reader in) { super(in); }
  public String readLine() throws IOException;
  public String readParagraph() throws IOException;
  public boolean isOver() throws IOException;
}
```

- How to test without the need to read a file?

Fake Object

Replace a real feature with an alternative implementation

Emulates real behavior, with characteristics customized to test case

it is not controlled nor observed by test case

Fake Object - Reasons

- Real component is not completed
- Real component is no longer available
- Real component is too slow
- Real component causes undesirable collateral effects

Double installation

- Once a double is created and configured, we need to tell the test to use it
- There are many ways to do that.

Dependency Injection

- Client (or test) sets dependencies in the unit under test
  - By setter
  - By parameter
  - By constructor

Dependency lookup

- Unit under test asks a specific component to create or find the object it needs
  - Factory of objects: create a real component
  - Service locator: get an object from an unknown place.

Advantages of using doubles

- Isolates unit test (*bug* in a unit does not affect test of another unit)
- Accelerates creation and execution of tests
- Allows to test even if a dependency is not completed or available
- Avoids undesirable collateral effects

The end

http://www.agilcoop.org.br

http://ccsl.ime.usp.br

http://qualipso.org

agilcoop@agilcoop.org.br