Software Patterns

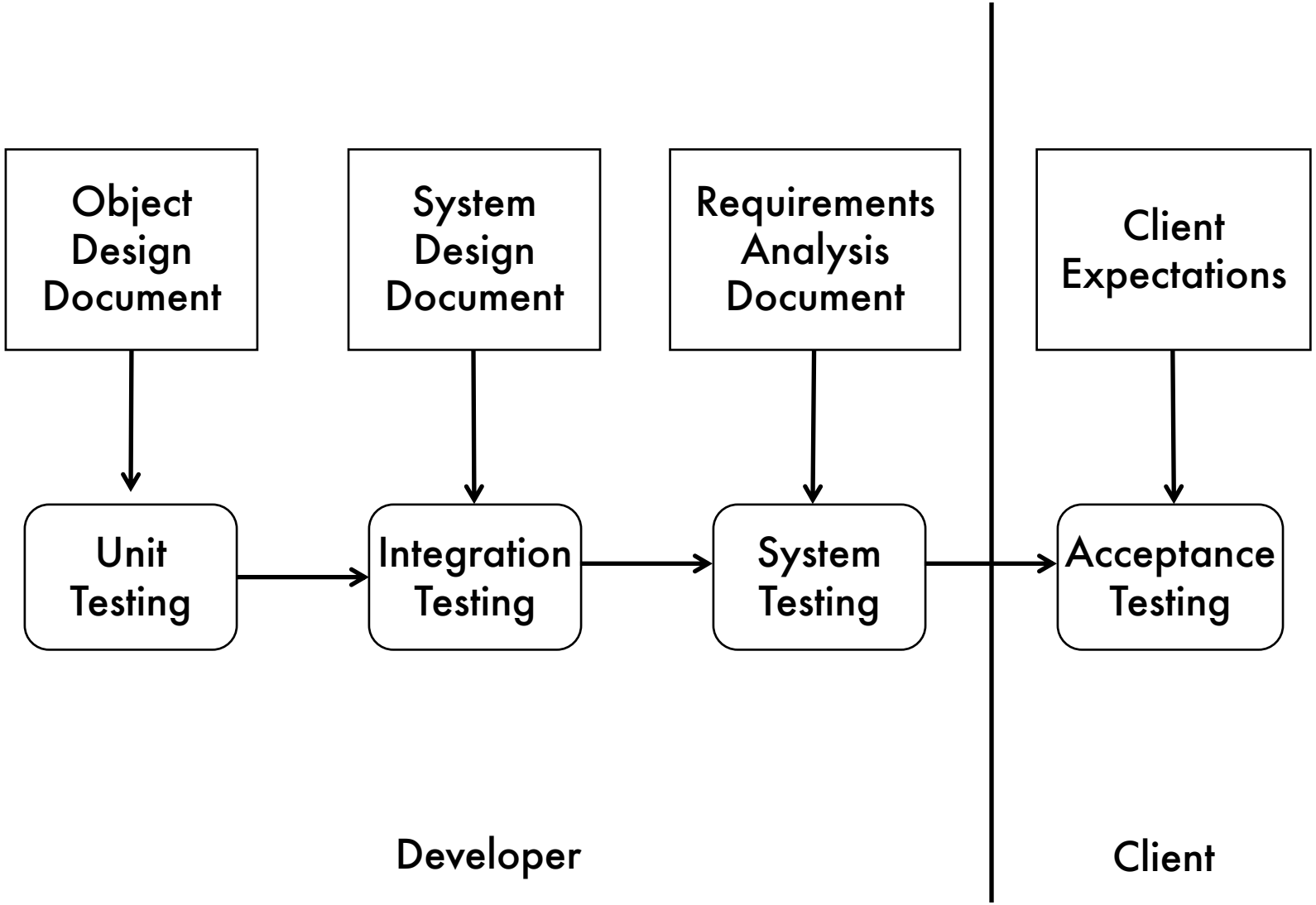
L07: Testing Patterns

Marlo Häring & Prof. W. Maalej (@maalejw)

Outline of the talk

1. Unit Testing
2. JUnit
3. Mock Objects
4. Dependency Injection

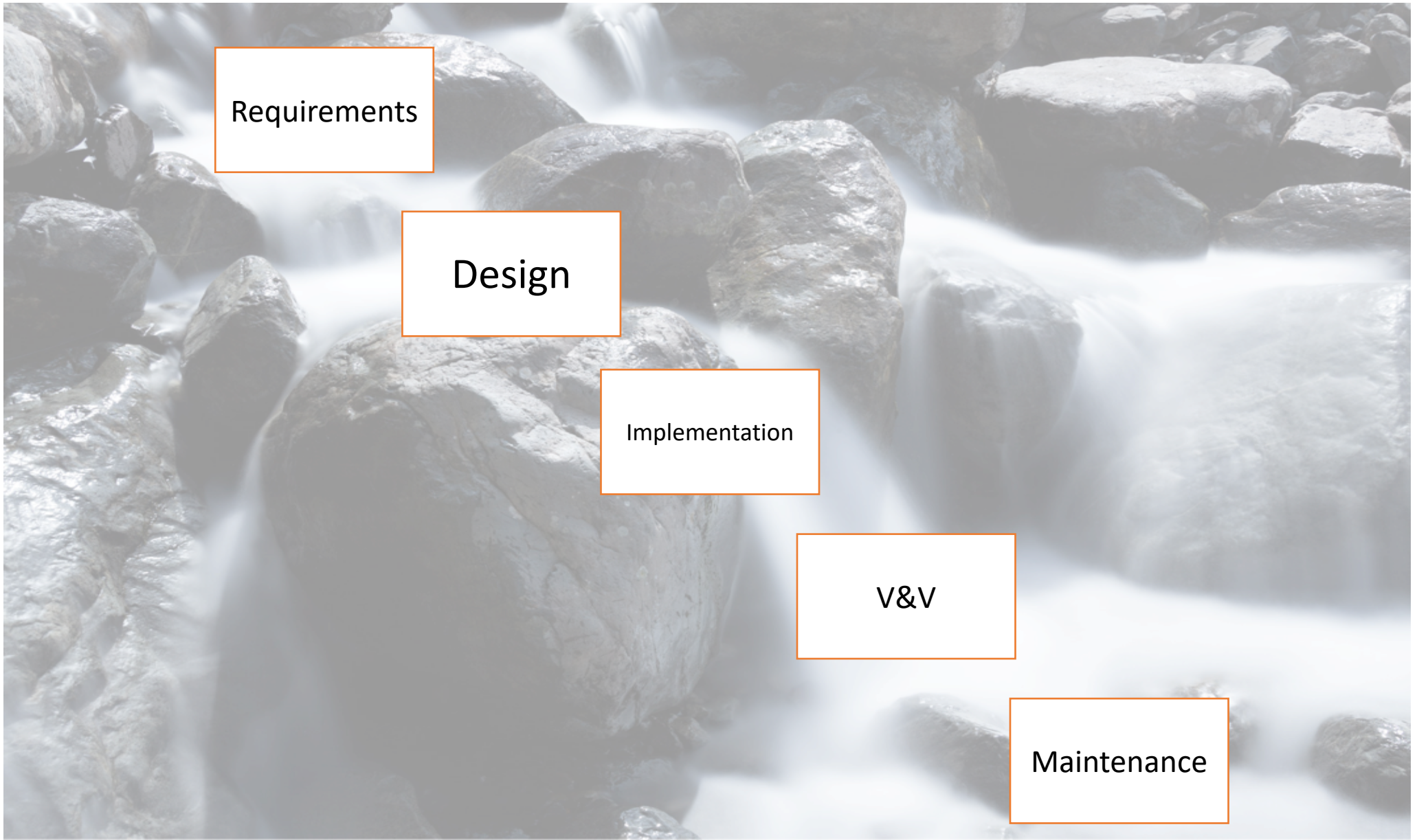
Testing activities



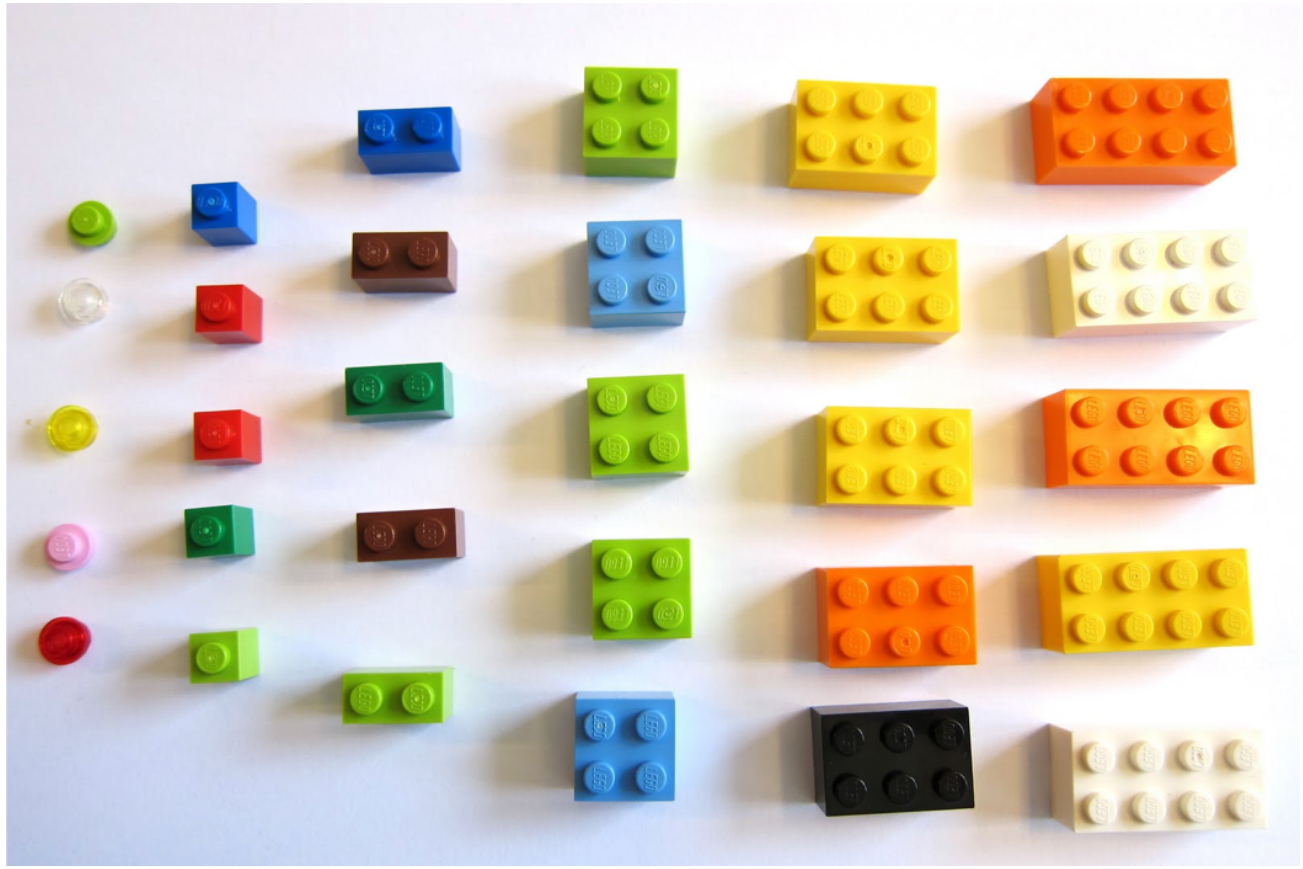
Types of testing

- Unit Testing
  - Individual component (class or subsystem)
  - Carried out by developers
  - Goal: Confirm that the unit is correctly implemented and behaving
- Integration Testing
  - Groups of subsystems (collection of subsystems) and eventually the entire system
  - Carried out by developers
  - Goal: Test the interfaces among the subsystems
- System Testing
  - The entire system
  - Carried out by developers
  - Goal: Determine if the system meets the requirements (functional and nonfunctional)
- Acceptance Testing
  - Evaluates the system delivered by developers
  - Carried out by clients. May involve typical transactions at the client site on a trial basis
  - Goal: Demonstrate that the system meets the requirements and is ready to use

Unit testing

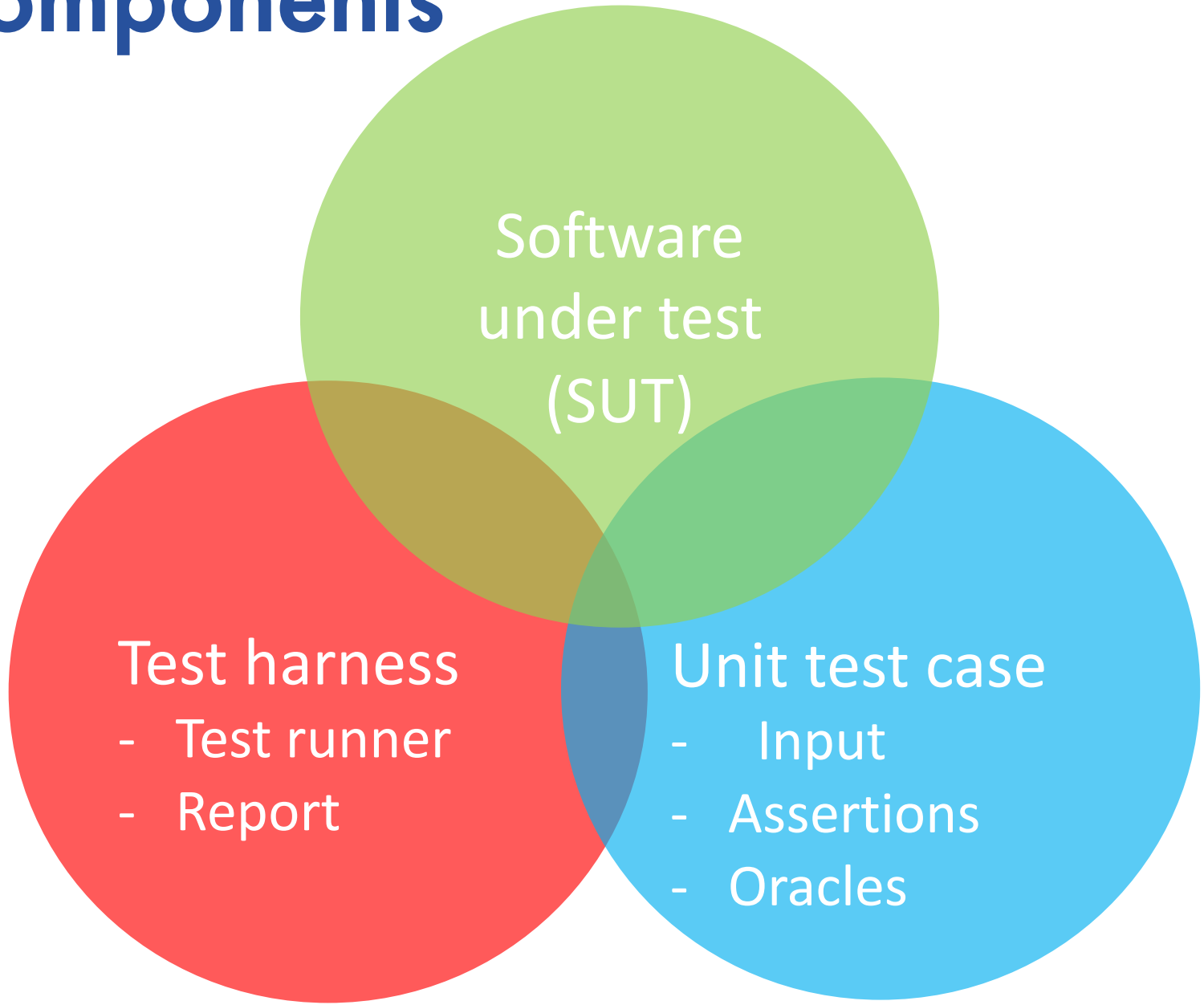


Unit



- Small chunk of code
  - Class/Interface
  - (public) Method
- Written by the same developers
- White-box
- Can be isolated

Testing components



Test model includes…

Software under test (SUT): The *unit* of interest

Test case: Description of the tests to be performed

- Often derived from scenarios and use cases
- Simply called test

- Input data needed for the SUT
- Oracle mechanism to decide whether the test is successful

Test harness

- A framework for running the tests under varying conditions and monitoring the behavior and outputs of the **system under test** (SUT)
- Necessary for automated testing

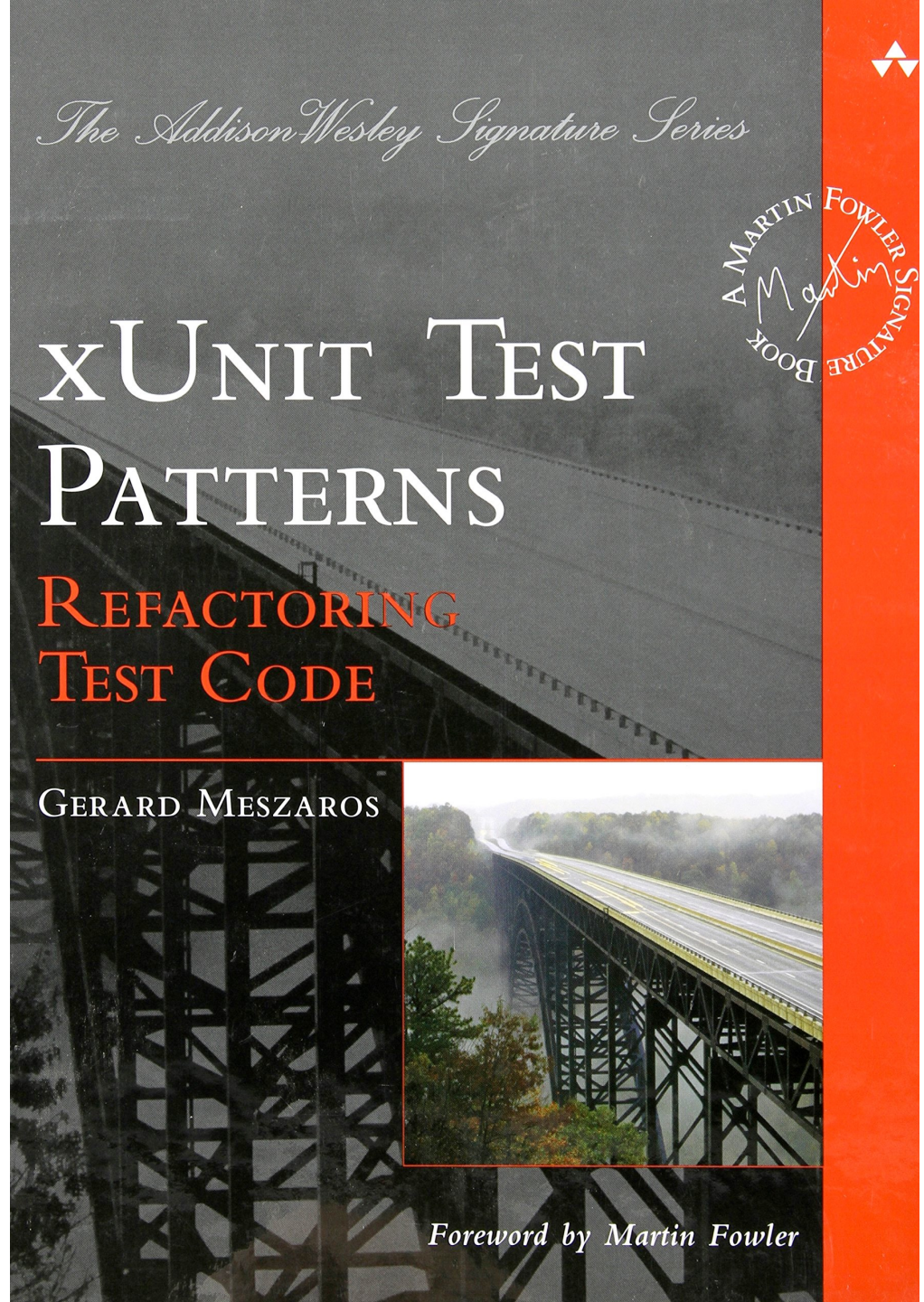
- Test driver: Programs executing one or more tests
- Reporting: Feedback about the status of the tests

F.I.R.S.T. tests

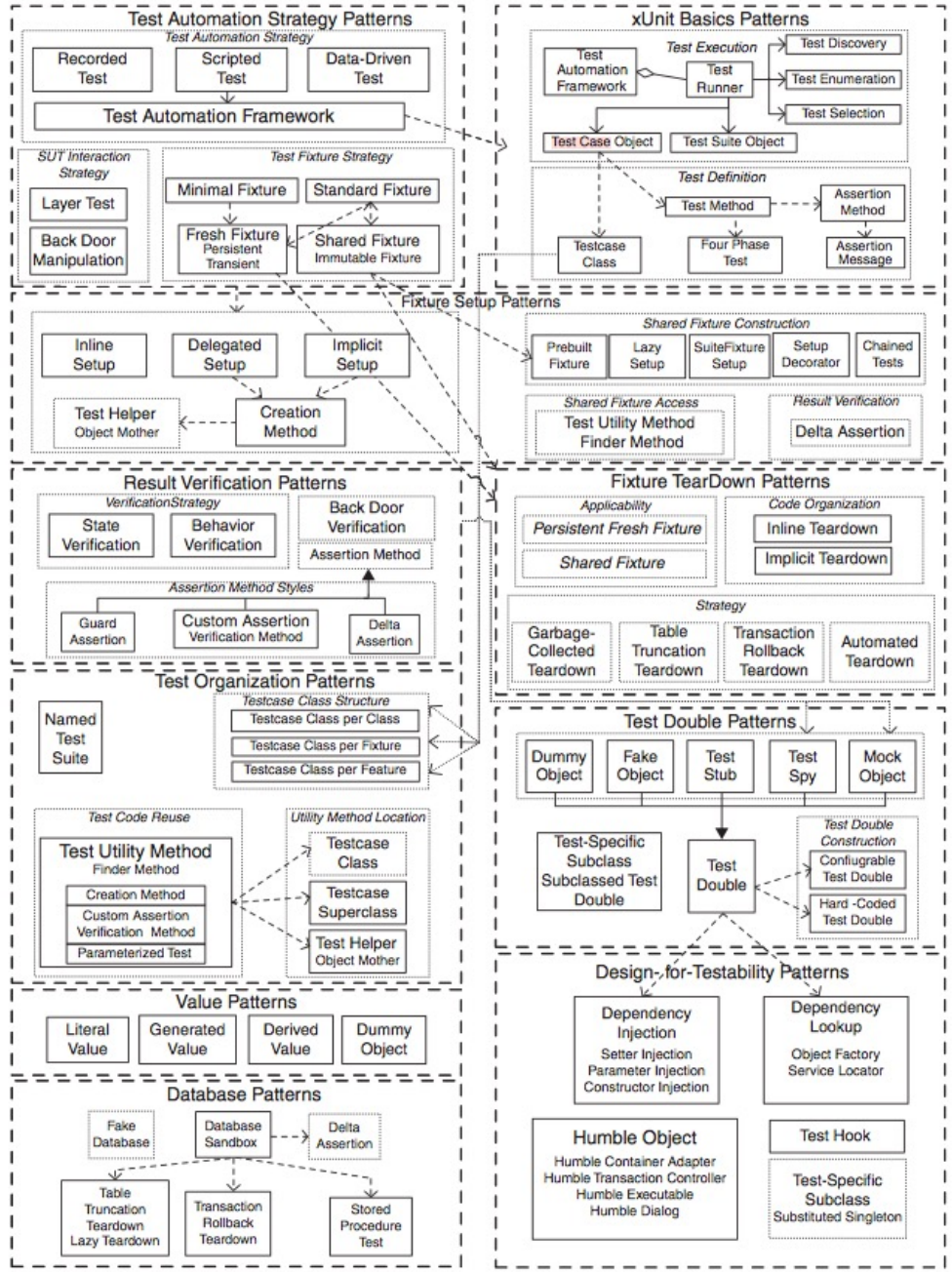
- Fast: Should take a finite amount of time to complete (few seconds to a minute).
- Independent: No order-of-run dependency.
- Repeatable: No dependency on external environment/instance in which it is running.
- Self-validating: No manual inspection required to check whether it has passed or failed.
- Thorough: Cover (every) use case scenario

AAA steps

- Arrange: set up the necessary preconditions and inputs to run SUT.
- Act: execute the SUT
- Assert: check the expected results.


The Addison Wesley Signature Series
A Martin Fowler Signature Book
xUnit Test Patterns
Refactoring Test Code
Gerard Meszaros
Foreword by Martin Fowler

The Patterns



Outline of the talk

1. Unit Testing
2. **JUnit**
3. Mock Objects
4. Dependency Injection

JUnit Overview

- A Java framework for writing and running unit tests
  - Test cases
  - Test suites
  - Test runner
- Simple but yet powerful
  - Written by Kent Beck and Erich Gamma on the Zurich-Atlanta flight when participating at OOPSLA'97
  - *"Never in the field of software development have so many owed so much to so few lines of code"* (M. Fowler)
  - Very elegant code base https://github.com/junit-team/junit4
- Actively developed under EPL
  - 143 contributors
  - > 7k stars

Unit testing example

```
class Money {
    private int fAmount;
    private String fCurrency;
    public Money(int amount, String currency) {
        fAmount= amount;
        fCurrency= currency;
    }
    public int amount() {
        return fAmount;
    }
    public String currency() {
        return fCurrency;
    }
```

```
    public Money add(Money m) {
        return new Money(amount()+m.amount(), currency());
    }
}
```

Unit testing `add()` with JUnit 4

The unit test MoneyTest tests whether the sum of two Moneys with the same currency contains a value that is the sum of the values of the two Moneys

```
import org.junit.Test;

public class MoneyTest {
   @Test public void simpleAdd() {
      Money m12CHF= new Money(12, "CHF");
      Money m14CHF= new Money(14, "CHF");
      Money expected= new Money(26, "CHF");
      Money observed= m12CHF.add(m14CHF);
   }
}
```

the name of the test case

Here we are calling the SUT Method add()

Unit Testing add() with JUnit 4

The unit test MoneyTest tests that the sum of two Moneys with the same currency contains a value that is the sum of the values of the two Moneys

```
import org.junit.Test;
import static org.junit.Assert.*;

public class MoneyTest {
    @Test public void simpleAdd() {
      Money m12CHF = new Money(12, "CHF");
      Money m14CHF = new Money(14, "CHF");
      Money observed = m12CHF.add(m14CHF);
      Money expected = new Money(26, "CHF");
      assertTrue(expected.equals(observed));
    }
}
```

Assertion: Returns True if parameter of type Boolean evaluates to True

@Ignore: omitting tests

- Sometimes certain tests should not be executed by the test harness, e.g.:
  - The current release of a third-party library used in the SUT has a bug in a routine Foo. We cannot test Bar until Foo is fixed

```
public class CalculatorTest {
    @Ignore("Don‘t run the Bar test until bug in Foo is fixed")
    @Test public void Bar() {
       ...
    }
}
```

@Test(timeout): making sure tests are short

- Unit tests should be short
- Some tests take long, particularly if network connectivity is involved
  - Here it is recommended to set an upper bound for the test

```
@Test(timeout=5000)
public void testNetworkOperation() {
  ...
}
```

@Before and @after: ensuring pre- and post conditions

Any Method can be decorated with @Before and @After:

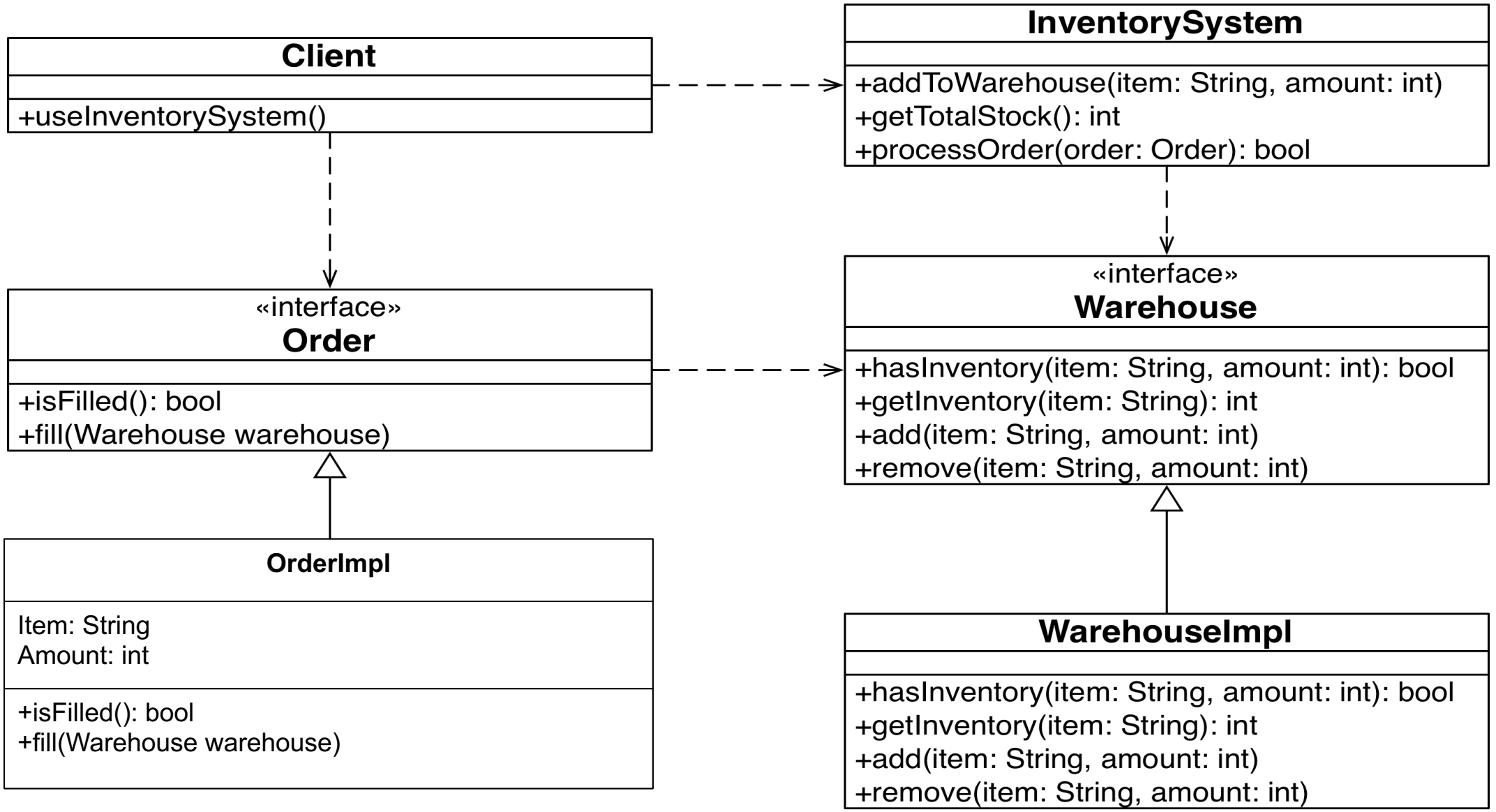
```
public class CalculatorTest {
  @Test public void addTest()
  @Test public void subTest()
   @Before public void setupTestData(){}//executed before every addTest/subTest
   @After public void teardownTestData(){}//executed after every addTest/subTest
}
```

Any Method can be decorated with @BeforeClass and @AfterClass

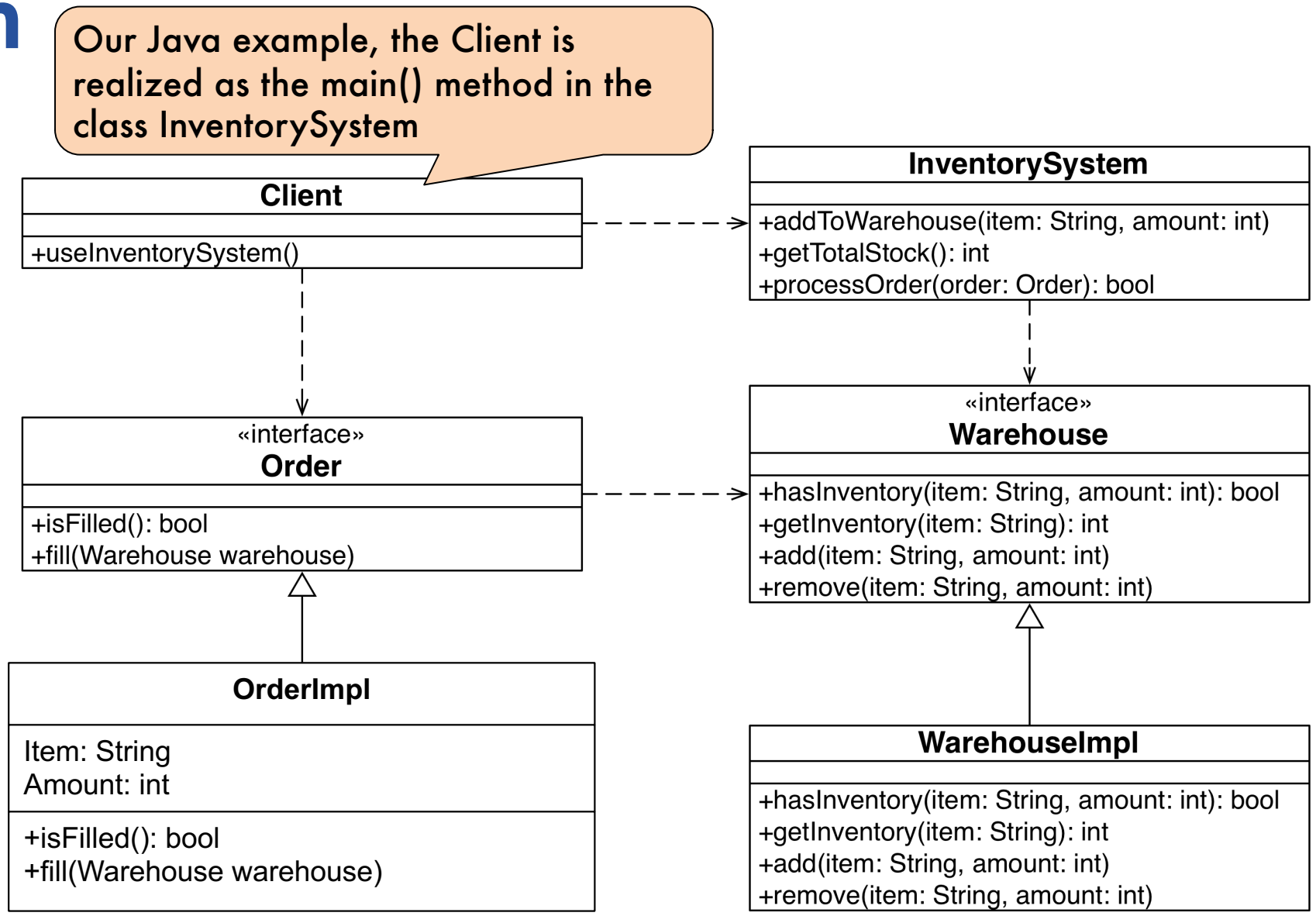
Useful for expensive setups that do not need to be run for every test

```
public class CalculatorTest {
  @BeforeClass // executed at instantiation of class
        public static void setupDatabase Connection() { ... }
  @AfterClass // executed after removing instance of class
        public static void teardownDatabase Connection() { ... }
 }
```

Example: model of a simple inventory system



Example: model of a simple inventory system



Java code for the Warehouse class

```
public interface Warehouse {
    public boolean hasInventory(String item, int amount);
    public int getInventory(String item);
    public void add(String item, int amount);
    public void remove(String item, int amount);
}
public class WarehouseImpl implements Warehouse {
    private Map<String, Integer> inventory = new HashMap<String, Integer>();

    public void add(String item, int amount) {
        inventory.put(item, amount);
    }
    public int getInventory(String item) {
        return inventory.get(item);
    }
    public boolean hasInventory(String item, int amount) {
        return inventory.get(item) >= amount;
    }
    public void remove(String item, int amount) {
        inventory.put(item, inventory.get(item) - amount);
    }
}
```

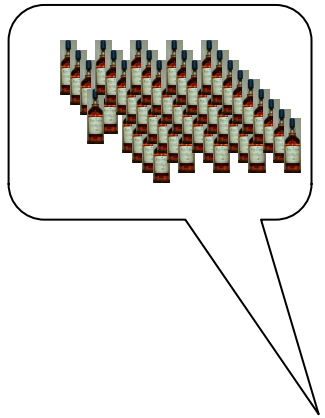
Java code for the Order class

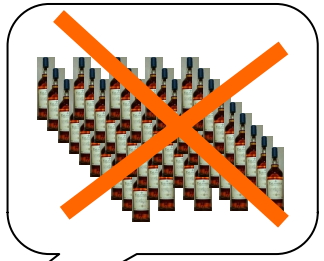
```
public interface Order {
    public boolean isFilled();
    public void fill(Warehouse warehouse);
}
```

```
public class OrderImpl implements Order {
    private String item;
    private int amount;
    private boolean filled = false;

    public OrderImpl(String item, int amount) {
        this.item = item;
        this.amount = amount;
    }
    public void fill(Warehouse warehouse) {
        if(warehouse.hasInventory(item, amount)) {
            warehouse.remove(item, amount);
            filled = true;
        }
    }
    public boolean isFilled() {
        return filled;
    }
}
```

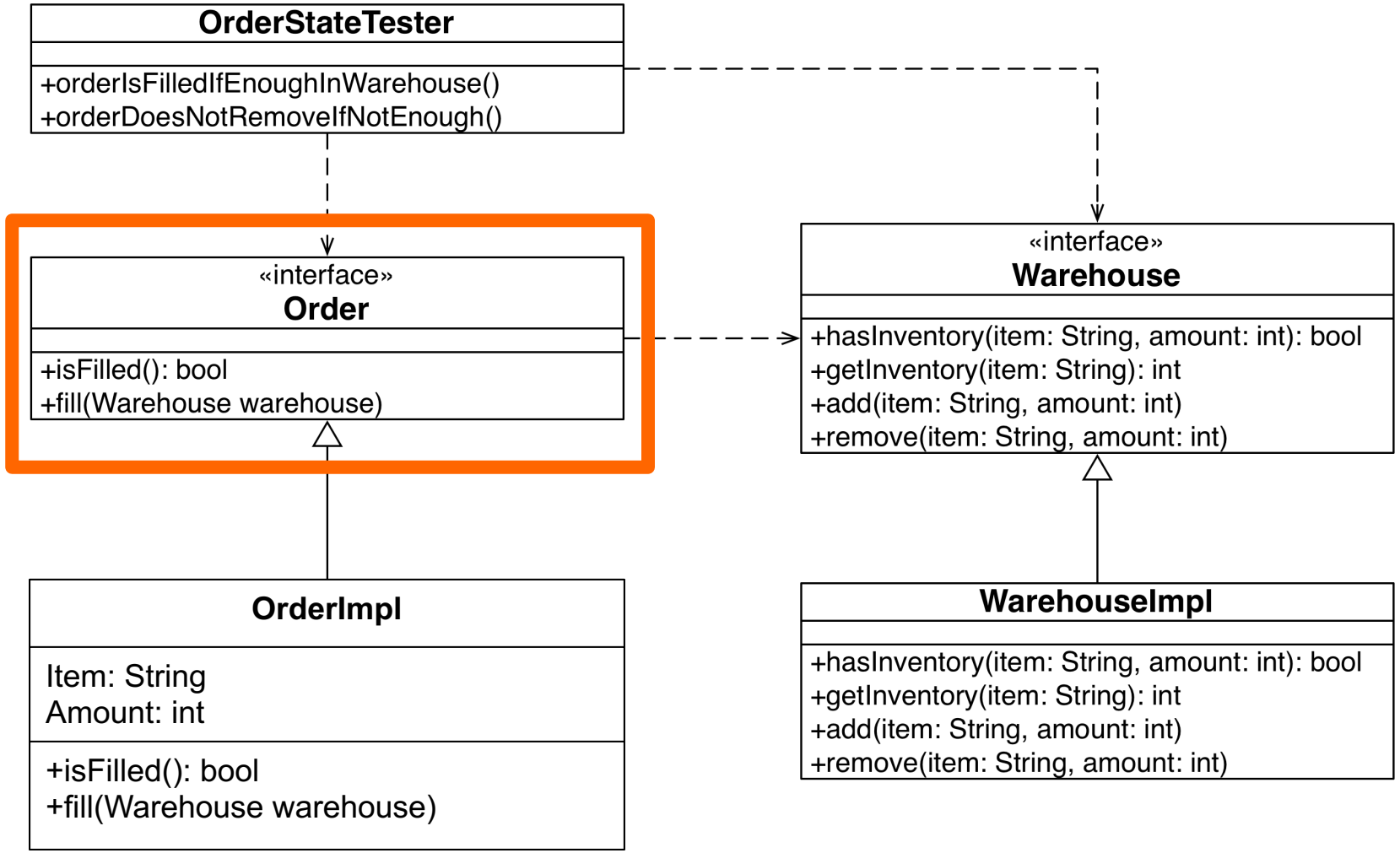
Java code for the InventorySystem class

```
public class InventorySystem {
   private static String TALISKER = "Talisker";
   private int totalStock;
   private Warehouse warehouse = new WarehouseImpl();
   public void addToWarehouse(String item, int amount) {
      warehouse.add(item, amount); totalStock += amount;
   }
   public int getTotalStock() {
      return totalStock;
   }
   public boolean processOrder(Order order) {
      order.fill(warehouse);
      return order.isFilled();
   }
   public static void main(String[] args) {
      InventorySystem inventorySystem = new InventorySystem();
      inventorySystem.addToWarehouse(TALISKER, 50);
      boolean order1success = inventorySystem.processOrder(new OrderImpl(TALISKER, 50));
      boolean order2success = inventorySystem.processOrder(new OrderImpl(TALISKER, 51));
      System.out.println("Order1 succeeded? " + order1success + " - Order2 succeeded? " + order2success);
   }
}
```



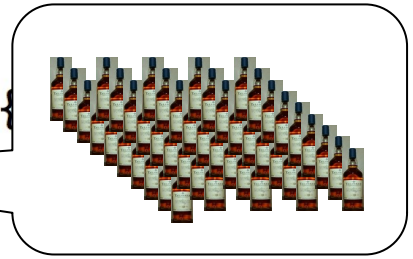


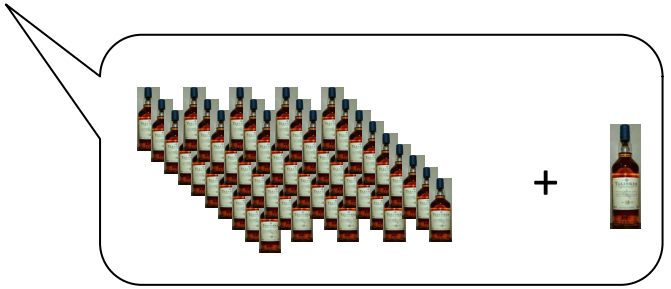
Unit-testing the inventory system



Order state tester class: A test suite for unit-testing the Order class

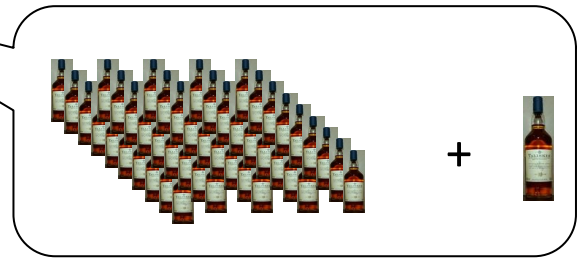
```
public class OrderStateTester {
    private static String TALISKER = "Talisker";
    private Warehouse warehouse = new WarehouseImpl();

    @Before public void setUp() throws Exception {
        warehouse.add(TALISKER, 50);
    }
    @Test public void orderIsFilledIfEnoughInWarehouse() {
        Order order = new Order(TALISKER, 50);
        order.fill(warehouse);
        assertTrue(order.isFilled());
    }
    @Test public void orderDoesNotRemoveIfNotEnough() {

                    ?

  }
}
```



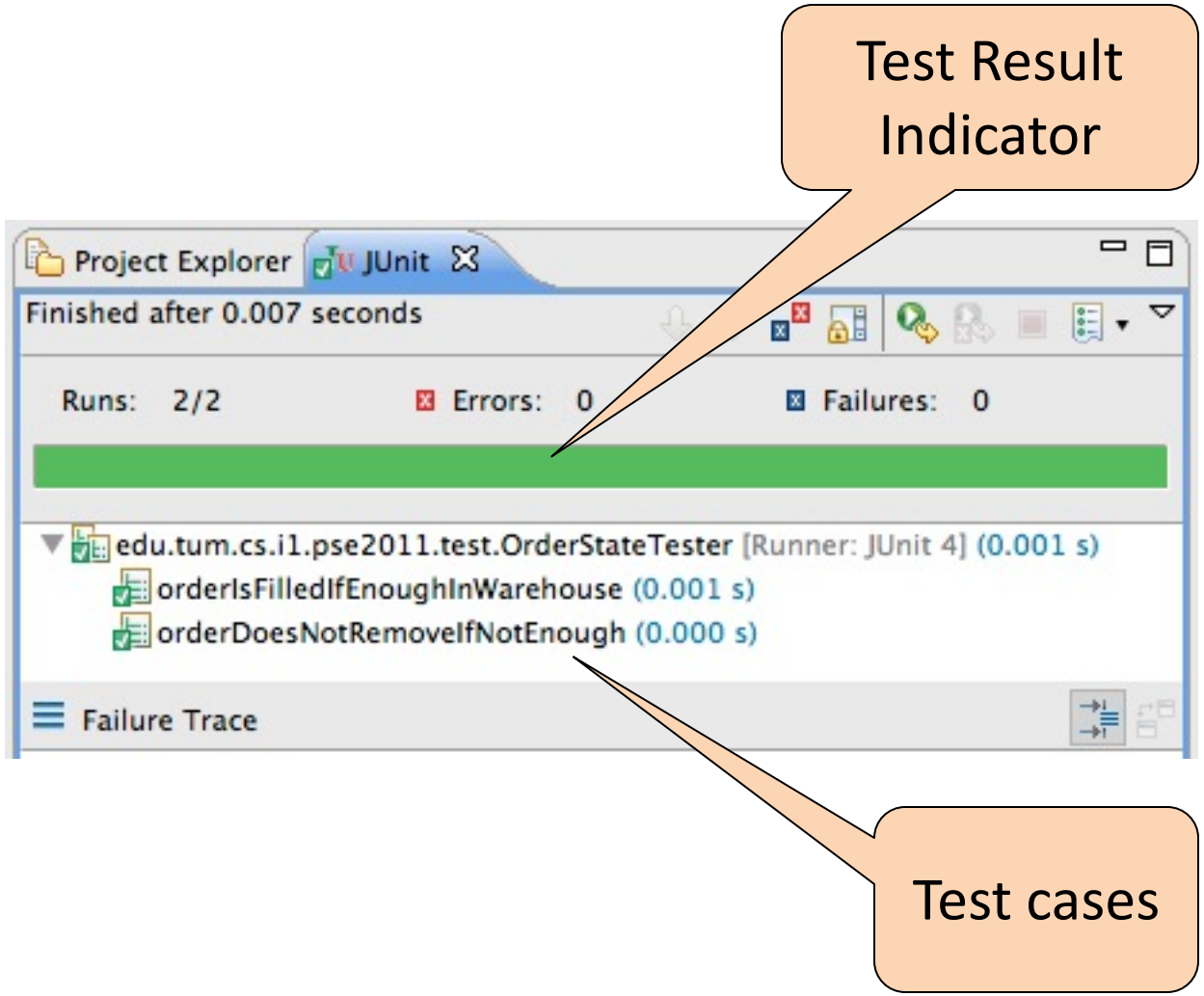


Order state test exercise: solution

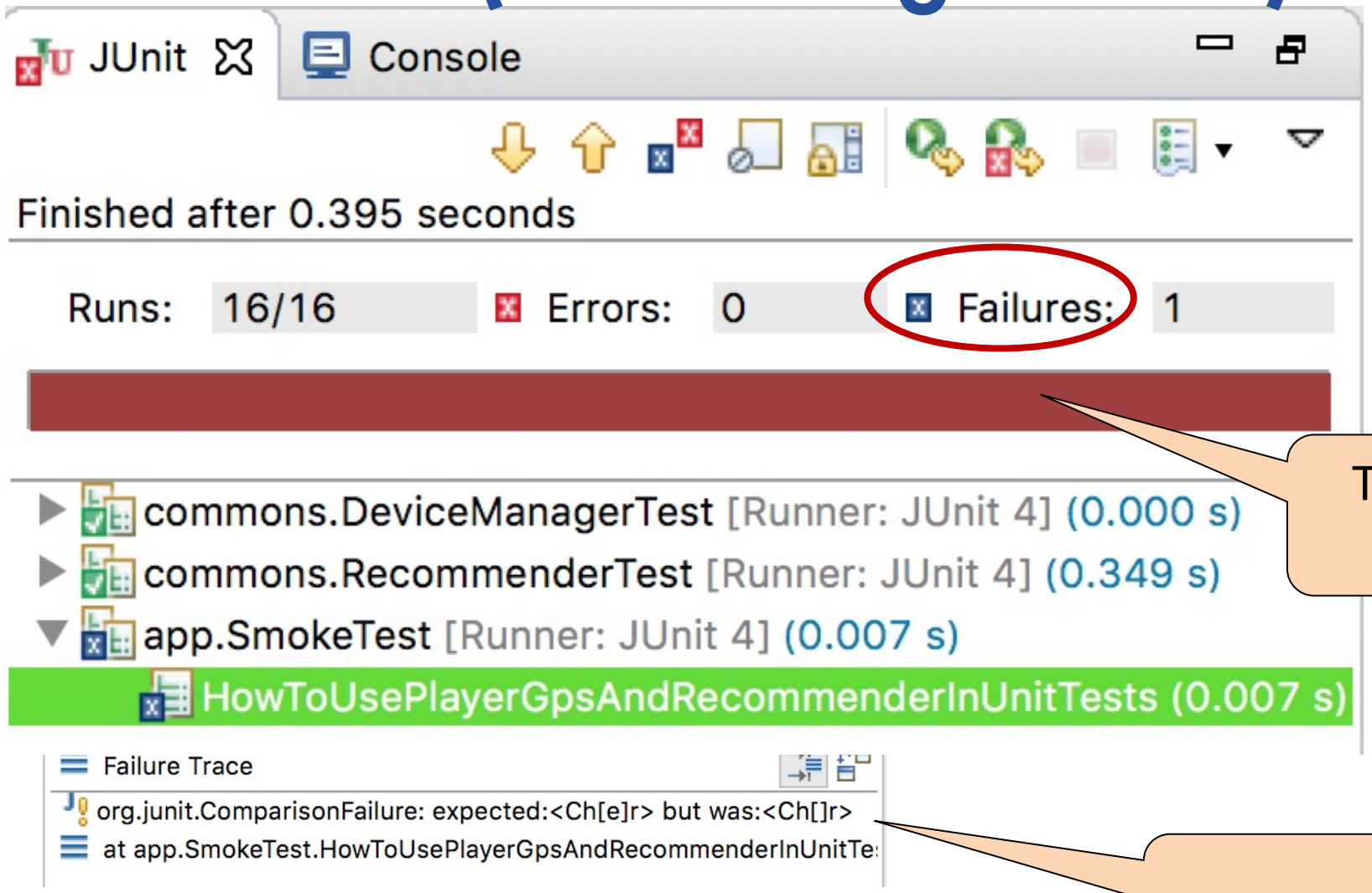
```
/**
 * This unit test case tests the state of Order
 * in case the warehouse does <b>not</b> have <b>enough</b>
 items in stock to fill the order.
 */
@Test
public void orderDoesNotRemoveIfNotEnough() {
  // Create a new order for 51 bottles Talisker.
  Order order = new OrderImpl(TALISKER, 51);
  // Try to fill the order.
  order.fill(warehouse);
  // Ensure that this order cannot be filled.
  assertFalse(order.isFilled());
}
```



Order state test exercise: JUnit output



TestRunner (a little digression)



Test Result Indicator

Expected vs. Actual

Testing the order class in 4 stages

```
public class OrderStateTester4S {
    private static String TALISKER = "Talisker";
    private Warehouse warehouse;

    @Before public void setUp() throws Exception {
        warehouse = new WarehouseImpl();
        warehouse.add(TALISKER, 50);
    }
    @After public void tearDown() throws Exception {
        warehouse = null;
    }
    @Test public void orderIsFilledIfEnoughInWarehouse() {
        Order order = new Order(TALISKER, 50);
        order.fill(warehouse);
        assertTrue(order.isFilled());
        assertEquals(0, warehouse.getInventory(TALISKER));
    }
    @Test public void orderDoesNotRemoveIfNotEnough() {
        ...
    }
}
```

1. A) Setup prerequisite objects:
   The warehouse

4. Tear down the prerequisite objects

1. B) Continued setup of prerequisite objects: Create new order

2. Run the test: Call the method being tested

3. Evaluate the results:
   - The order must be filled
   - The warehouse must be empty

Exception testing

- Sometimes the *expected* behavior of the SUT is to raise a (checked) Exception (e.g., *InvalidOrderAmountException)*
  - In Java *MyOwnException extends java.lang.Exception*
- JUnit can verify such behavior with an *ad-hoc* annotation
  - @Test(expected = MyOwnException.class)

Order Exception test

```
public void fill(Warehouse warehouse) throws  InvalidOrderAmountException {
            if(warehouse.hasInventory(item, amount)) {
                  warehouse.remove(item, amount);
                  filled = true;
            } else throw new InvalidOrderAmountException();
 }


@Test(exception=InvalidOrderAmountException.class)
public void orderDoesNotRemoveIfNotEnough() throws   InvalidOrderAmountException {
            Order order = new OrderImpl(TALISKER, 51);
            order.fill(warehouse);
}
```

From state testing to behavior testing

- JUnit helps us to test the state of a SUT
- What if we not only want to test the state of a SUT, but also its interaction with its collaborators:
  - E.g., interaction between Order and Warehouse?
- Limitation of JUnit:
  - JUnit does not provide mechanisms to test that a specific sequence of operations takes place (in a defined order)
  - Which operations are called on collaborators and in which order

Behavior-based testing promotes testability

Testability

"Degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met."

(IEEE 610.12)

Testability

In order to obtain *Testability* we need:

- Observability
- Controllability

Complexity due to dependencies

The curse of dependencies

Dependencies are:

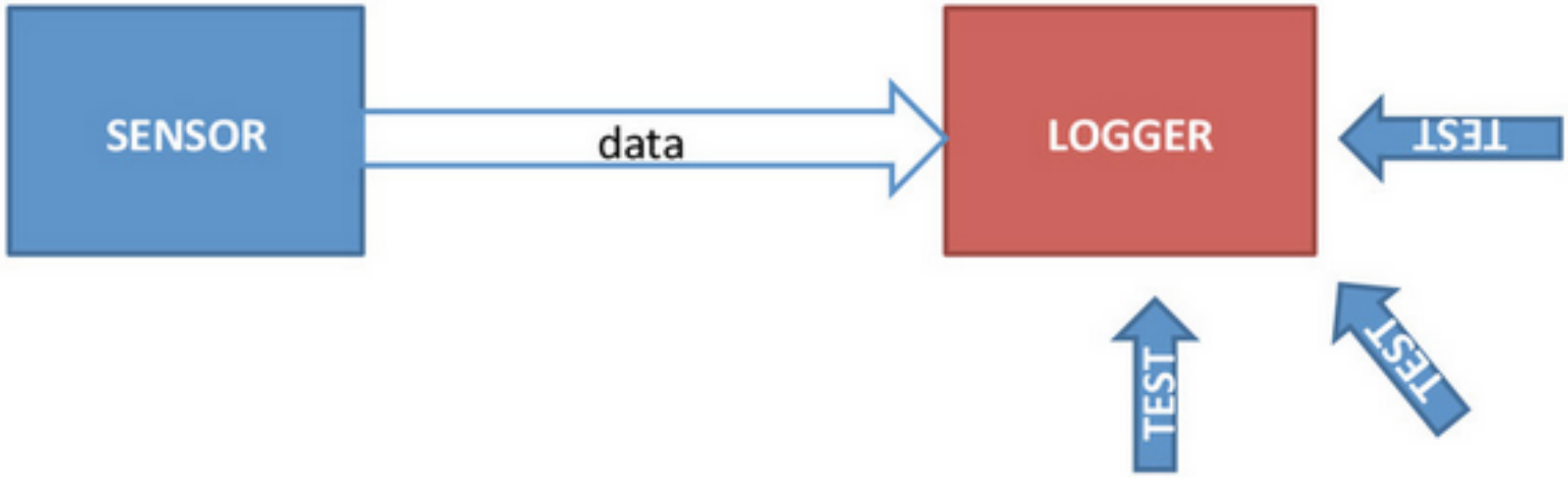
- Difficult to observe (black-box)
- Difficult to setup
- Slow
- Interleaved with other resources
- Plain and simply unreachable

Solution: Test doubles and mocks

Outline of the talk

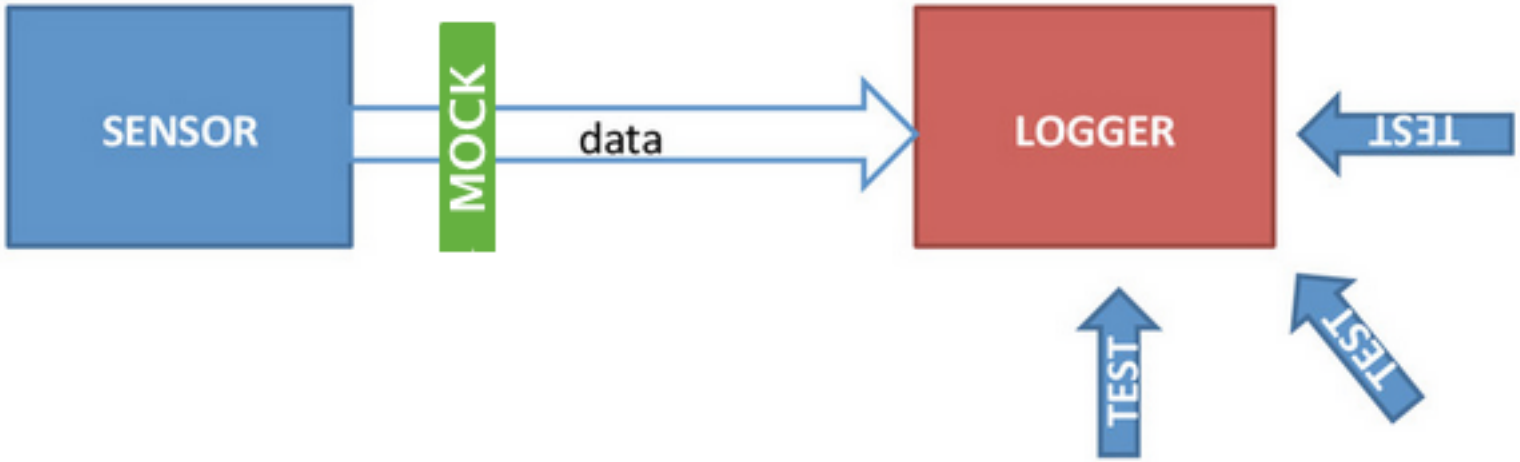
1. Unit Testing
2. JUnit
3. Mock Objects
4. Dependency Injection

Basic scenario for test doubles



- Sensor is the dependency (e.g., direct access to thermostat)
- Sensor sends data over the Logger (the SUT)
- In order to work, Logger needs its *collaborator* Sensor

Basic scenario for test doubles



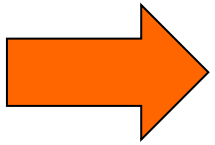
- Create a version of Sensor which can be
  - *controlled*
  - *observed*
- The Mock (or any other test double) is not a sensor, but behaves like one

Test doubles

- The name comes from the movies:
  - The "stunt double" replaces the movie actor, when it becomes dangerous
  - A "test double" generic name of objects or procedures that behave like the collaborator and simplify (or remove) dependencies when testing

Taxonomy of test doubles

- **Dummy object:** Often used to fill parameter lists, passed around but never used
- **Fake object:** A working implementation that contains a "shortcut" which makes it not suitable for production code
  - Example: A database stored in memory instead on a disk
- **Stub:** Provides canned answers to calls made during the test
  - Provides always the same answer
  - Example: Random number generator that always return 42

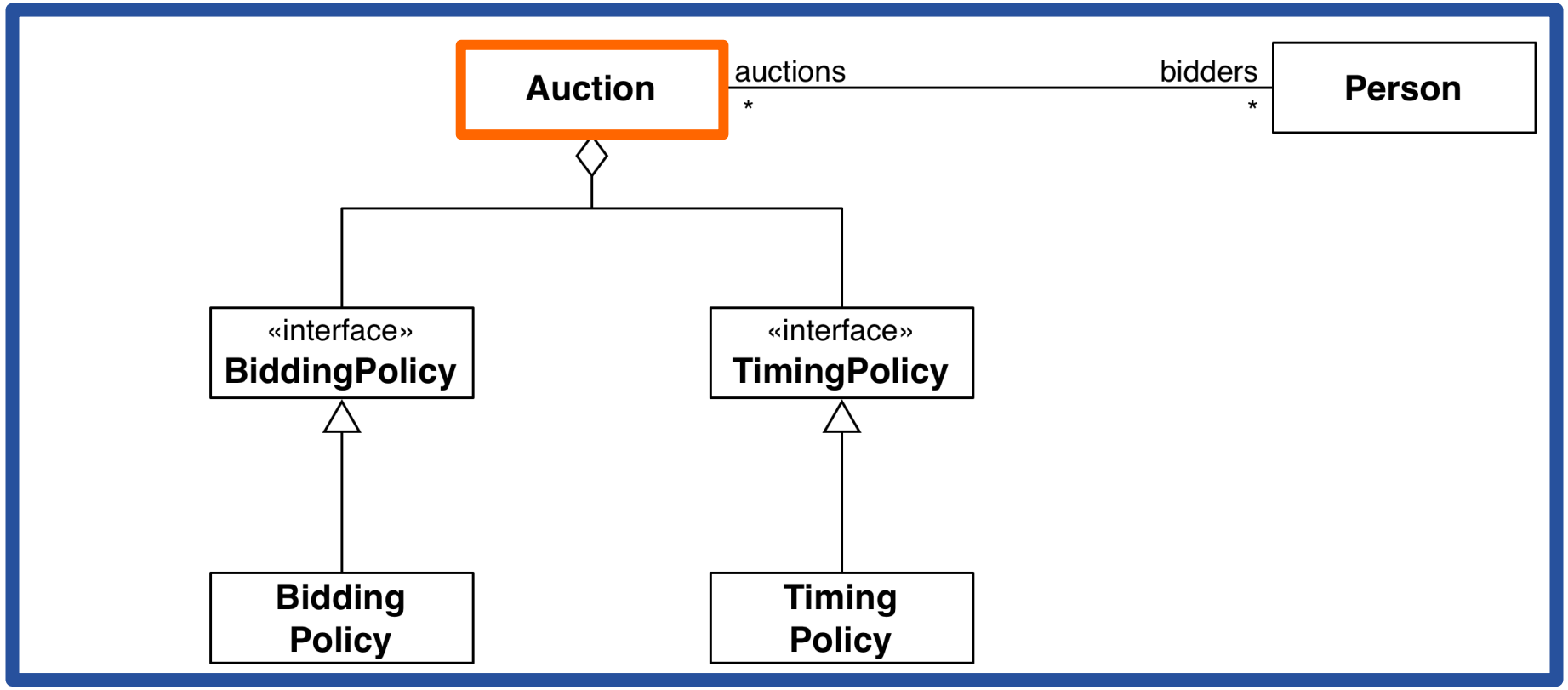


- **Mock object:** able to mimic the behavior of the real object
  - know how to deal with sequence of calls they are expected to receive
- **Test Spy**: able to mimic only *some* of the dependency (partial mocking)
  - when in doubt, use mocks

Good design is crucial when using mock objects:
Real object must be specified with an interface (façade)
and implementation or specification inheritance

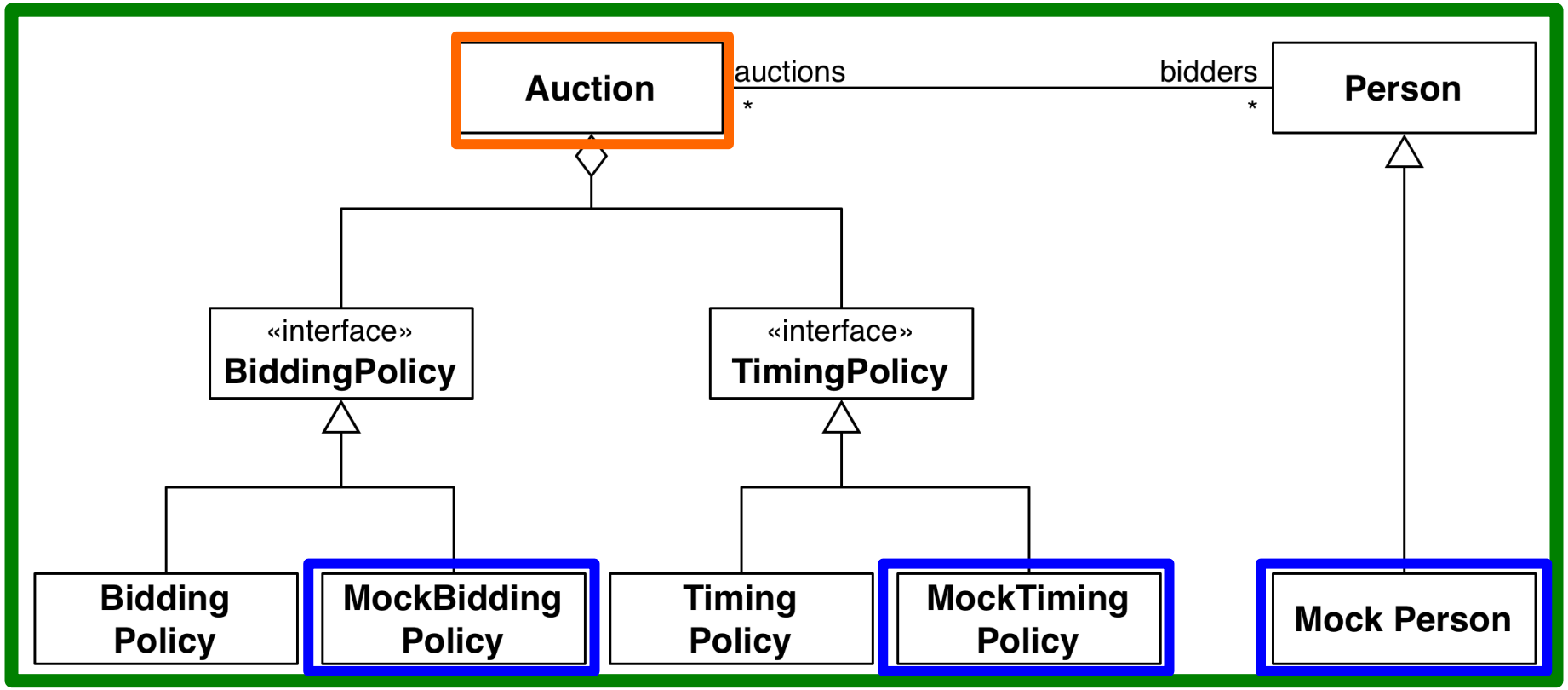
Motivation for mock objects

- Assume a system model for an auction system with 2 types of policies
- We want to unit test Auction, which is our SUT

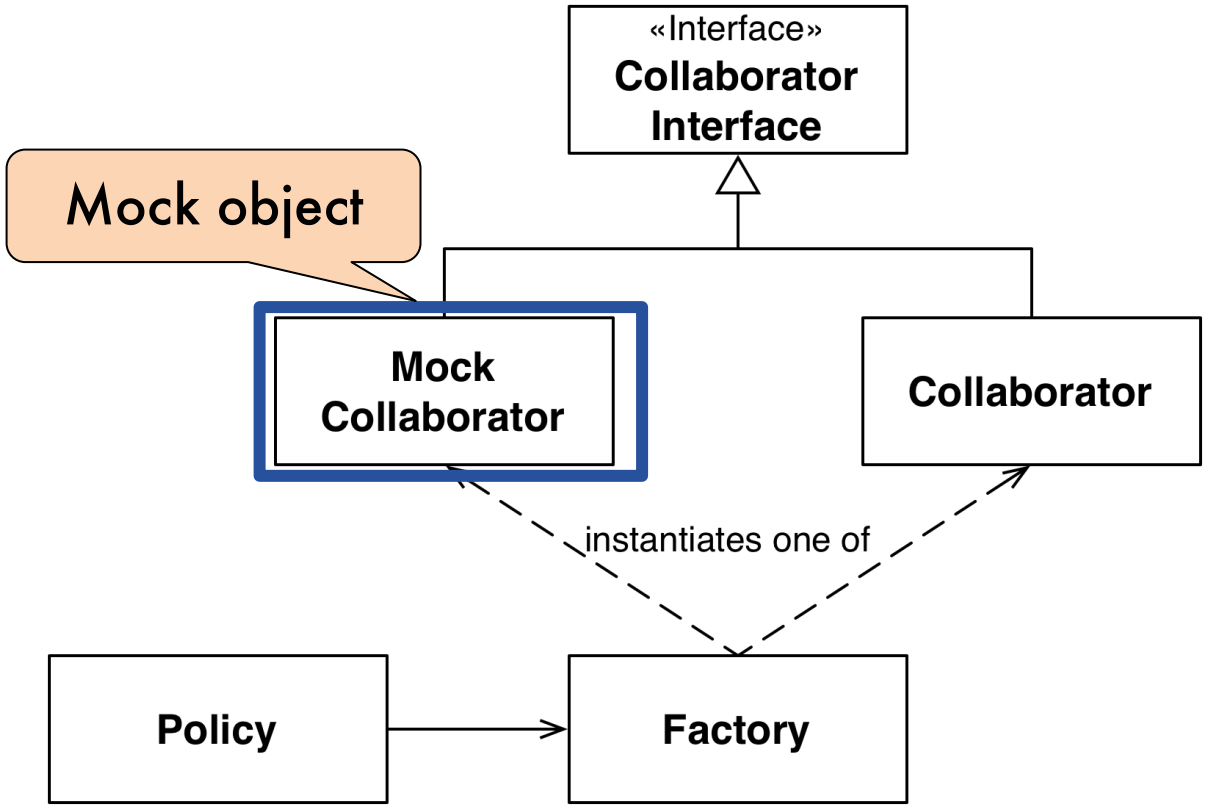


Motivation for mock objects

- The mock object pattern replaces the interaction with the collaborators in the system model (that is Person, the Bidding Policy and the TimingPolicy by mock objects)
- These mock objects can be created at startup-time with a factory pattern (by the framework)



Mock-object pattern



- In this pattern a mock object **replaces** the ***behavior*** of a real object called the **collaborator** and returns **hard-coded values**
- A mock object can be created at **startup-time** with a **factory** pattern
- Mock objects can be used for testing ***state*** *of individual objects* and the ***interaction*** *between objects*
- Validate that the interactions of the SUT with its collaborators **behave as expected**
- The selection of the Collaborator or Mock Collaborator is called ***binding***

Lifecycle of mock objects

- Create an **instance** of the mock object and **bind** it to the abstract class or interface
- Set the **state** in the mock object
  - Set any parameters or attributes that could be used by the object to test
- Set **expectations** in the mock object
  - Desired or expected outcome is specified (e.g., number of calls, including the number of method calls and the return values of mock object invocations)
- Put the mock in **ready state** so that the Mock Object knows that from now on it can start behaving *like* the real object but with expectation set above
- **Invoke SUT** using the Mock Object as its collaborator
- **Validate consistency** in the Mock Object
  - Implement *assert(s)* which verifies the state (using the values provided by the Mock Object)
  - Execute the *verify* method to check the expected behavior (function calls)

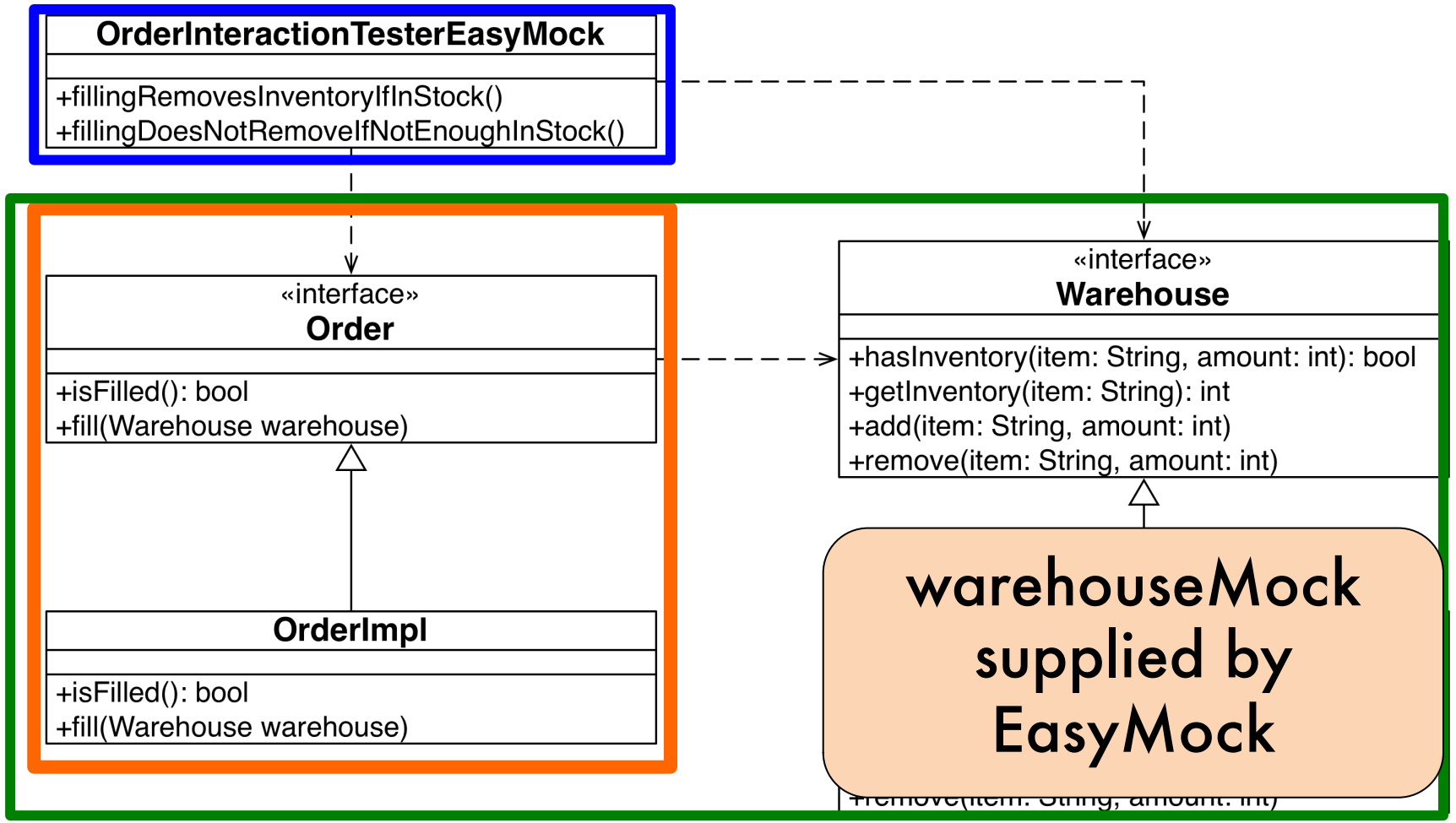
- Source: Brown & Tapolcsanyi: Mock Object Patterns. In Proceedings of the 10th Conference on Pattern Languages of Programs, 2003. http://hillside.net/plop/plop2003/papers.html

The EasyMock framework

- The goal: make it easy to use mock objects with unit testing framework
  - EasyMock uses static method imports (`org.easymock.EasyMock.*`)
  - It uses a record/replay metaphor
- Five steps to use the EasyMock framework:
  1. Create the mock object for the interface of the collaborator, e.g.:
     `mock = createMock(Interface.class)`
  2. Specify the expected behavior of the mock object:
     - Method returns type T:
       `expect(T methodCall).andReturn(T returnValue)`
     - Method returns void: `methodCall`
     - Method throws an exception: `.andThrow(E Exception)`
  3. Execute the mock object ("ready to play"): `replay(mock)`
  4. Invoke SUT methods
  5. Validate behavior (compares expected behavior with observed behavior), e.g.:
     `verify(mock)`

http://easymock.org/

Test model of InventorySystem with a mock object



Comparing expected and observed behavior with easy mock

```
public class OrderInteractionTesterEasyMock {
    private static String TALISKER = "Talisker";
    private Warehouse warehouseMock;

    @Test
    public void fillingRemovesInventoryIfInStock() {
        Order order = new OrderImpl(TALISKER, 50);
        warehouseMock = createMock(Warehouse.class);
        expect(warehouseMock
            .hasInventory(TALISKER, 50)).andReturn(true);
        warehouseMock.remove(TALISKER, 50);
        replay(warehouseMock);
        order.fill(warehouseMock);
        assertTrue(order.isFilled());
        verify(warehouseMock);
    }
}
```

1. Create mock object

2. Specify expected behavior ("behavioral oracle")

3. Set mock object to be ready to play

4. Validate observed behavior against expected behavior

Types of mocks in EasyMock

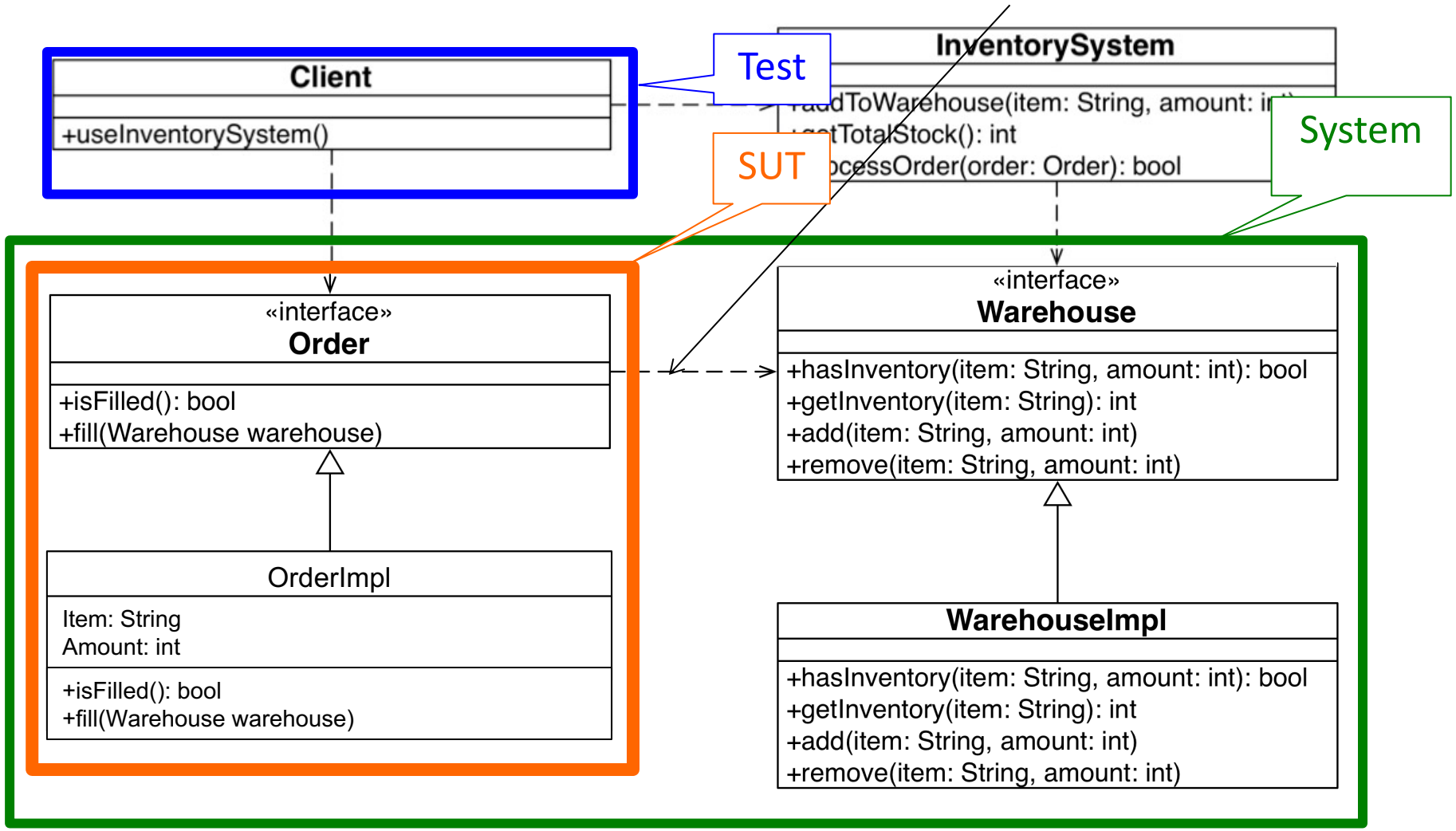
- Normal: created using `mock()` does not check the **order** of methods calls
- Strict: created using `strictMock()` checks the order of method calls
- Nice: created using `niceMock()` return a falsy value rather than `AssertionError` if an unexpected method is called.

Applicability of the mock object pattern

- Need to test an object with **non-deterministic** behavior
  - E.g. Current weather temperature
  - Violation of good test design: Repeatability
- An object is **difficult to set up**
  - E.g.: The set up takes too long for running many unit tests
  - Violation of good test design: Fast performance
- A specific behavior is **hard to trigger**
  - E.g.: network error
- The methods of an object are **very slow**
  - E.g.: Climate modeling
- The object has a **user interface** or is the user interface itself
  - The unit test needs to confirm that a callback function was actually called
- The real object cannot be tested, because it **does not exist** (yet)
  - Subsystem built by other teams or when interfacing with new hardware systems

Unit testing the order class

- We unit tested the class Order, our SUT, with OrderStateTester
- Problem: Order is not unit testable in isolation, it depends on Warehouse



From state testing to behavior testing

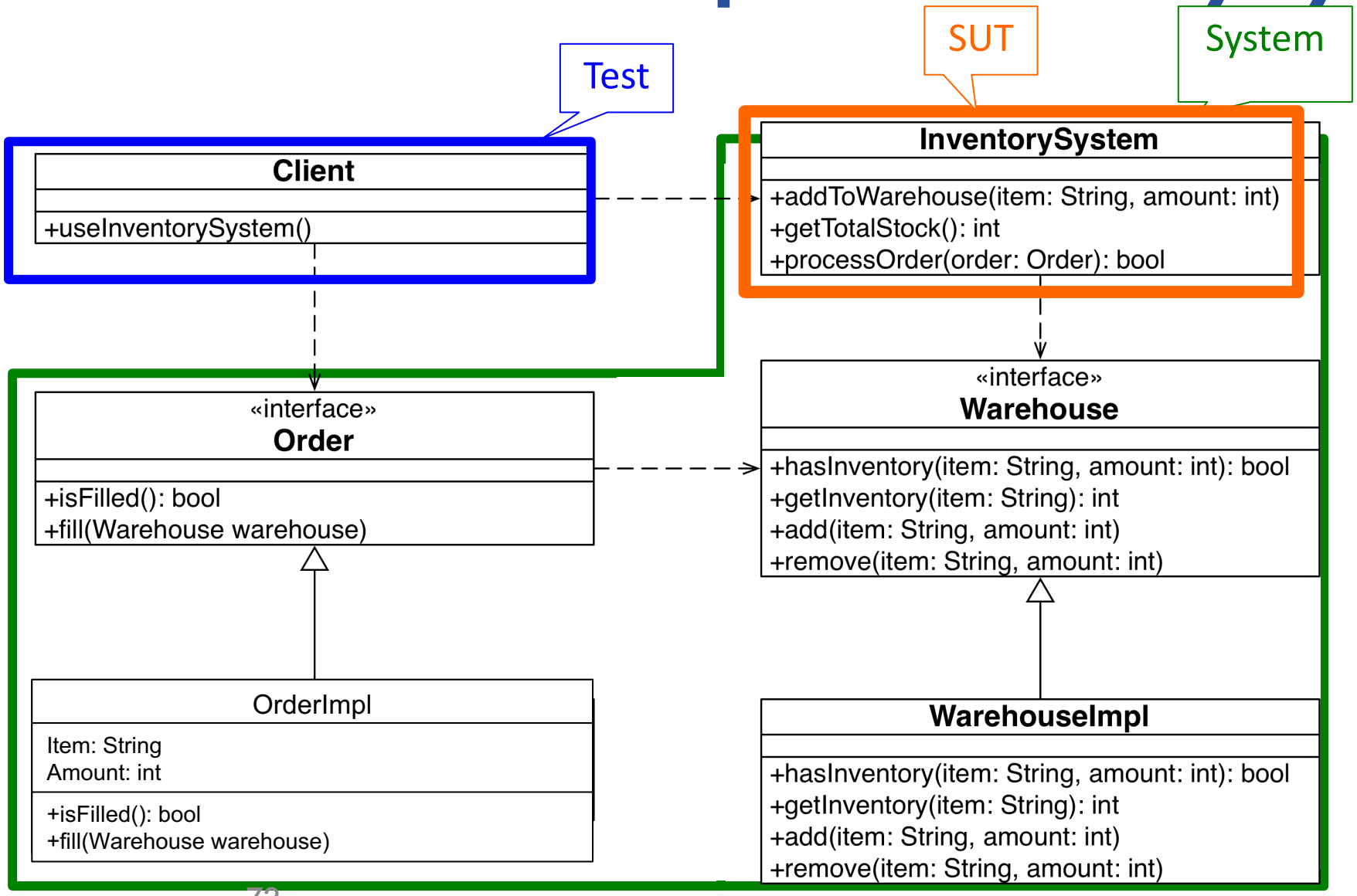
- **Observation 1**: JUnit helps us to test the state of a SUT
- Limitation:
  - No mechanisms to test a specific behavior
  - which operations are called on collaborators, in which order
- **Observation 2**: mock objects help to test behavior
- Limitation:
  - Mock objects create a **high coupling** between SUT and the rest of the system model (often not yet tested)

Reduce this coupling as much as possible with dependency injection

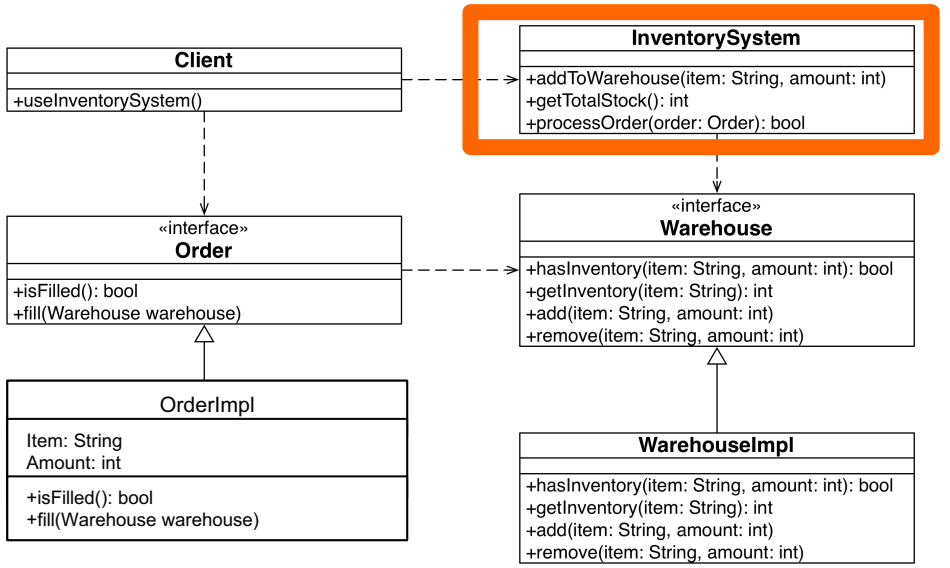
Outline of the talk

1. Unit Testing
2. JUnit
3. Mock Objects
4. Dependency Injection

UML model for the simple InventorySystem



Problem: high coupling during testing



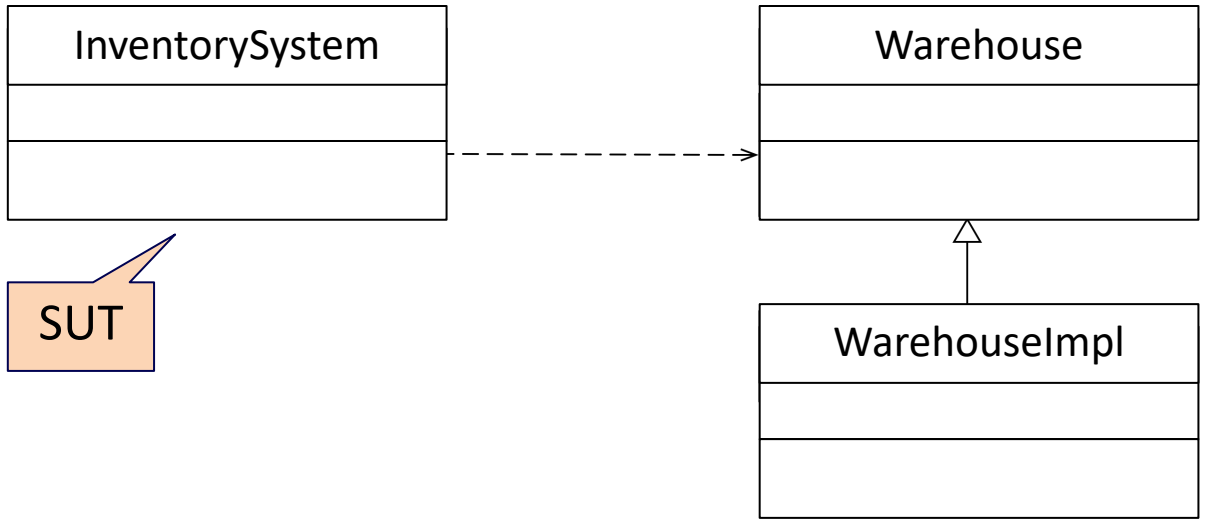
- InventorySystem

```
public class InventorySystem {
    private static String TALISKER = "Talisker“;
    private int totalStock;
    private Warehouse warehouse = new WarehouseImpl();
    …
}
```

- Since InventorySystem explicitly calls `new` to instantiate Warehouse, we cannot test this implementation using a mock Warehouse ☹
- It is impossible to switch the Warehouse to the mocked object without modifying
- Can we unit test InventorySystem with both, a production and a mock Warehouse while avoiding modification of InventorySystem itself?

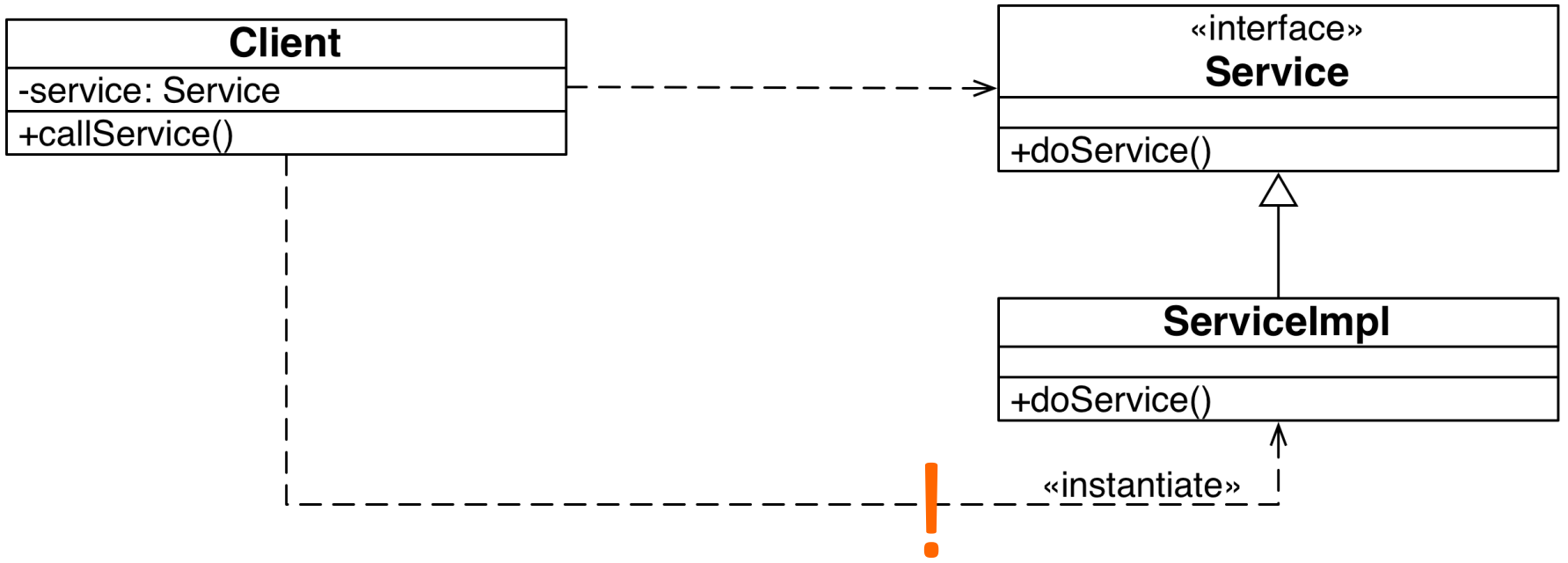
Instantiation with `new`

- What is wrong with the usual way of instantiating objects with new?
- Let's discuss this issue using Client-Server pattern



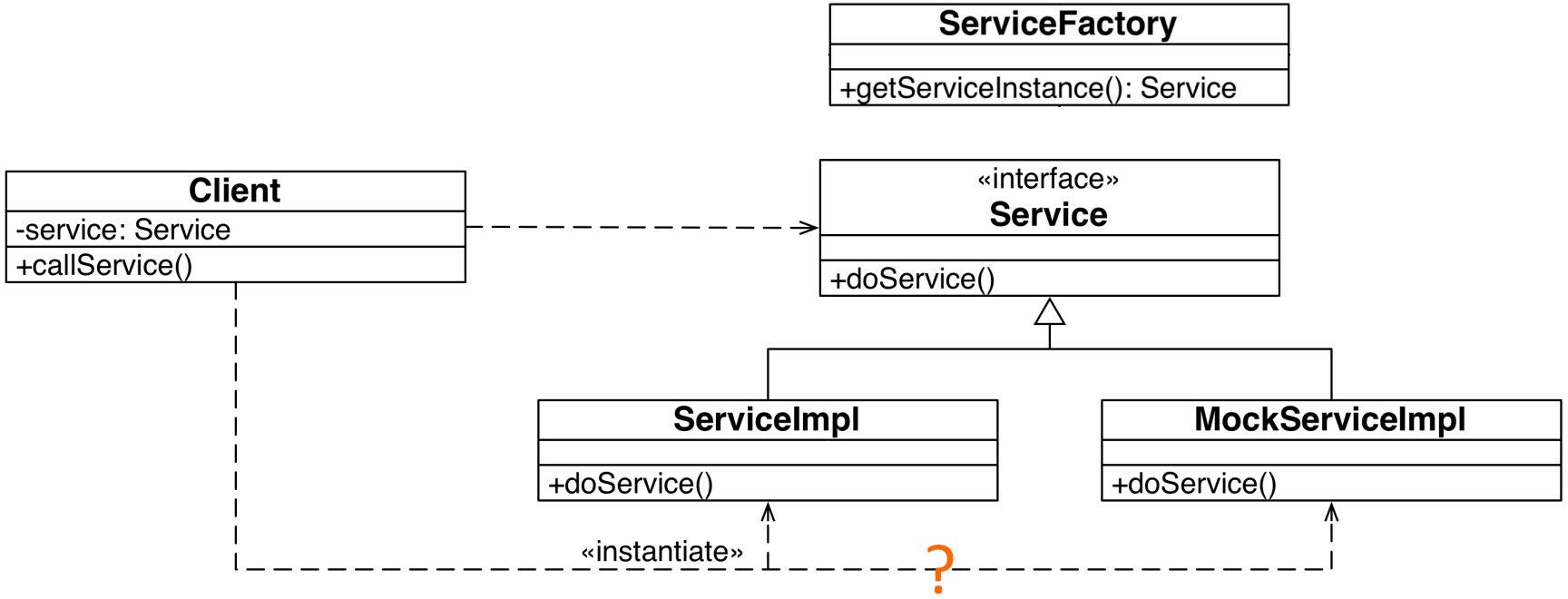
Instantiation with `new` (2)

- `new` creates a direct compile time dependency on the implementation

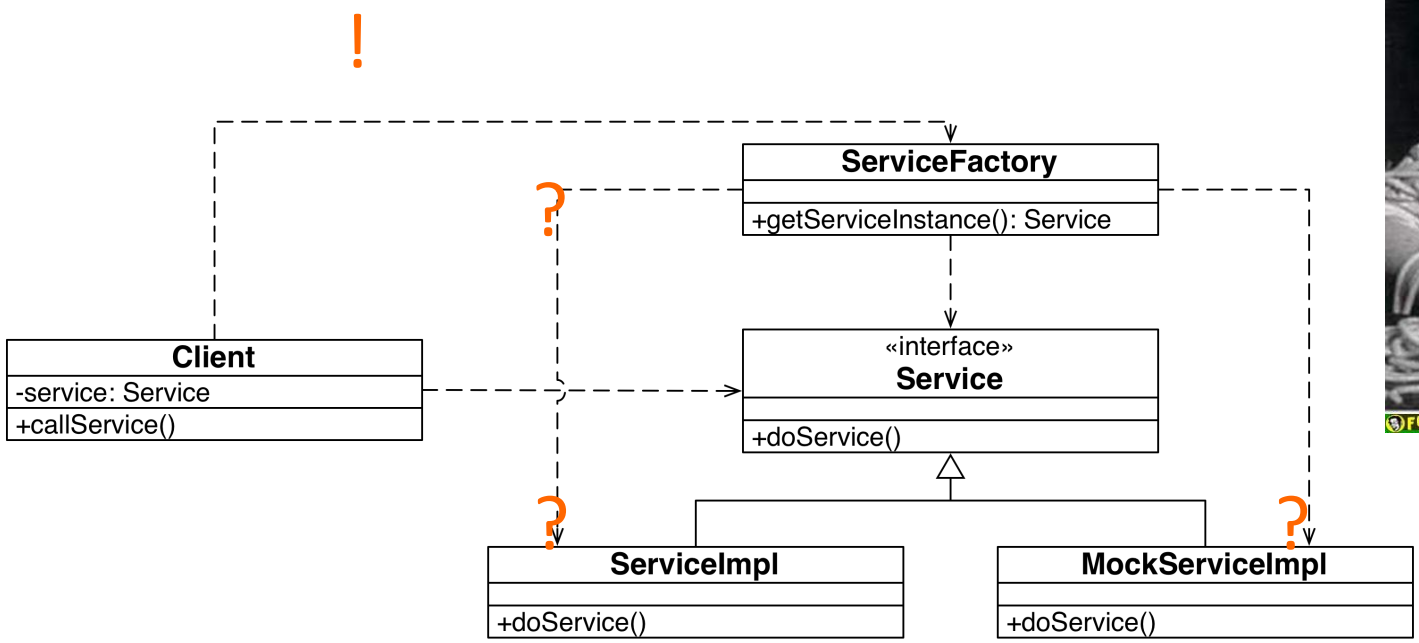


Instantiation with `new` (3)

- We cannot switch to a mock object without changing the Client code (our SUT)
- What about using a Factory?



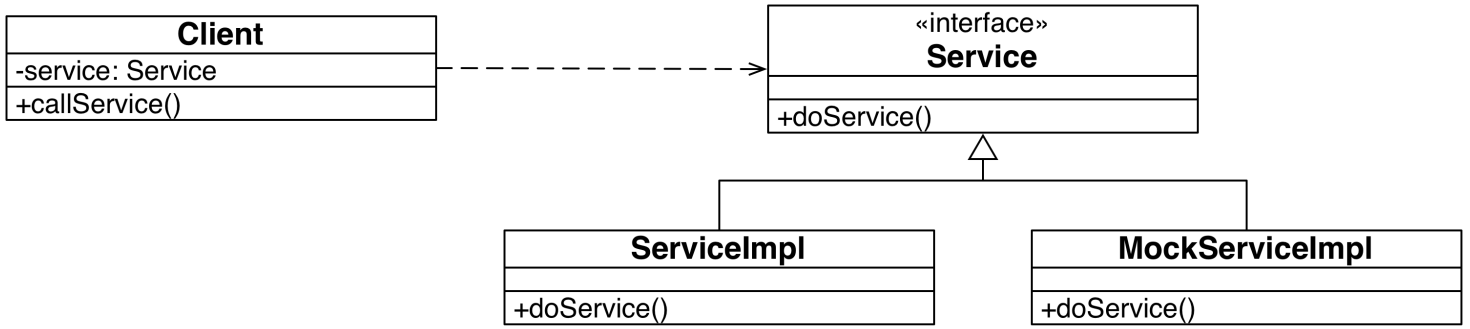
Instantiation with factories



- 4 associations that increase the coupling between objects
- 3 too many!

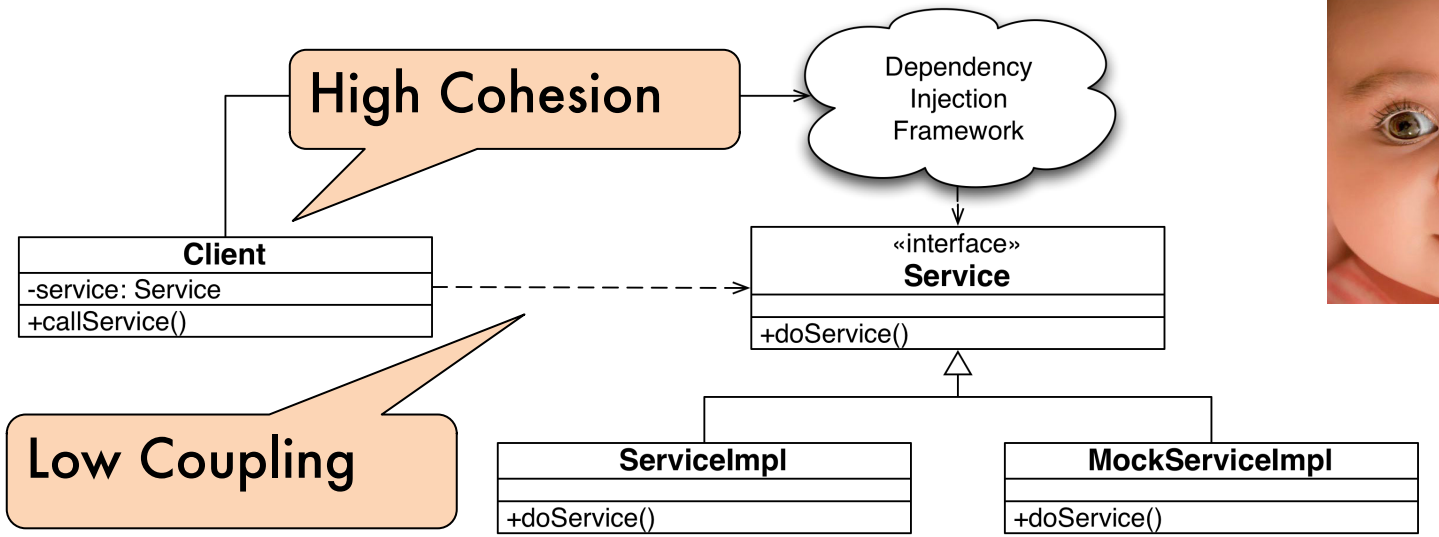
Introduction to dependency injection

- This is what we want:



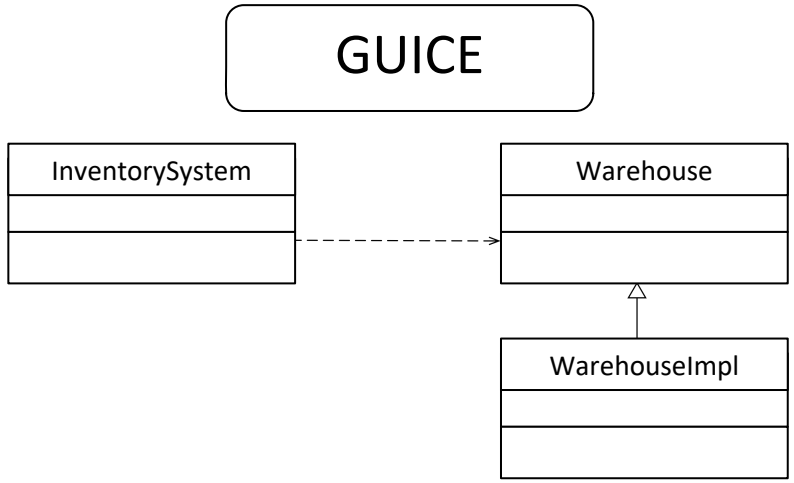
- Client only depends on the interface Service
- We want a mechanism that allows us to switch between different kinds of services easily
- Result: Low coupling!

Dependency injection reduces the coupling



- The Client only knows the Service interface
  - The implementations are injected by the framework
  - Binding (the selection of the "Impl" subclass to be instantiated for a certain interface) is done by the framework
  - There are no compile-time dependencies to implementations anymore
  - Because there is no factory, the Client class can be more easily reused

What is an injection?



How do you get the
Jam into the Krapfen?

The “Krapfen”:
InventorySystem

Injector filled with “Jam”
(WarehouseImpl)

Inventory system without Guice

```
public class InventorySystem {
   private static String TALISKER = "Talisker";
   private int totalStock;
   private Warehouse warehouse = new WarehouseImpl();
   public void addToWarehouse(String item, int amount) {
      warehouse.add(item, amount); totalStock += amount;
   }
   public int getTotalStock() {
      return totalStock;
   }
   public boolean processOrder(Order order) {
      order.fill(warehouse);
      return order.isFilled();
   }
   public static void main(String[] args) {

      InventorySystem inventorySystem = new InventorySystem();
      inventorySystem.addToWarehouse(TALISKER, 50);
      boolean order1success = inventorySystem.processOrder(new OrderImpl(TALISKER, 50));
      boolean order2success = inventorySystem.processOrder(new OrderImpl(TALISKER, 51));
      System.out.println("Order1 succeeded? " + order1success +
          " - Order2 succeeded? " + order2success);
   }
}
```

Inventory system with Guice

```
public class InventorySystemDI {
   private static String TALISKER = "Talisker";
   private int totalStock;
   @Inject private Warehouse warehouse;
   public void addToWarehouse(String item, int amount) {
      warehouse.add(item, amount); totalStock += amount;
   }
   public int getTotalStock() {
      return totalStock;
   }
   public boolean processOrder(Order order) {
      order.fill(warehouse);
      return order.isFilled();
   }
   public static void main(String[] args) {
      Injector injector = Guice.createInjector(new ProductionModule());
      InventorySystemDI inventorySystem = injector.getInstance(InventorySystemDI.class);
      inventorySystem.addToWarehouse(TALISKER, 50);
      boolean order1success = inventorySystem.processOrder(new OrderImpl(TALISKER, 50));
      boolean order2success = inventorySystem.processOrder(new OrderImpl(TALISKER, 51));
      System.out.println("Order1 succeeded? " + order1success +
          " - Order2 succeeded? " + order2success);
   }
}
```

Can you spot the difference?

Can you spot the difference?

```
public class InventorySystemDI {
    private static String TALISKER = "Talisker";
    private int totalStock;
    @Inject private Warehouse warehouse;
    public void addToWarehouse(String item, int amount) {
        warehouse.add(item, amount); totalStock += amount;
    }
    public int getTotalStock() {
        return totalStock;
    }
    public boolean processOrder(Order order) {
        order.fill(warehouse);
        return order.isFilled();
    }
    public static void main(String[] args) {
        Injector injector = Guice.createInjector(new ProductionModule());
        InventorySystemDI inventorySystem = injector.getInstance(InventorySystemDI.class);
        inventorySystem.addToWarehouse(TALISKER, 50);
        boolean order1success = inventorySystem.processOrder(new OrderImpl(TALISKER, 50));
        boolean order2success = inventorySystem.processOrder(new OrderImpl(TALISKER, 51));
        System.out.println("Order1 succeeded? " + order1success +
            " - Order2 succeeded? " + order2success);
    }
}
```

Notice there is no "new" here anymore!

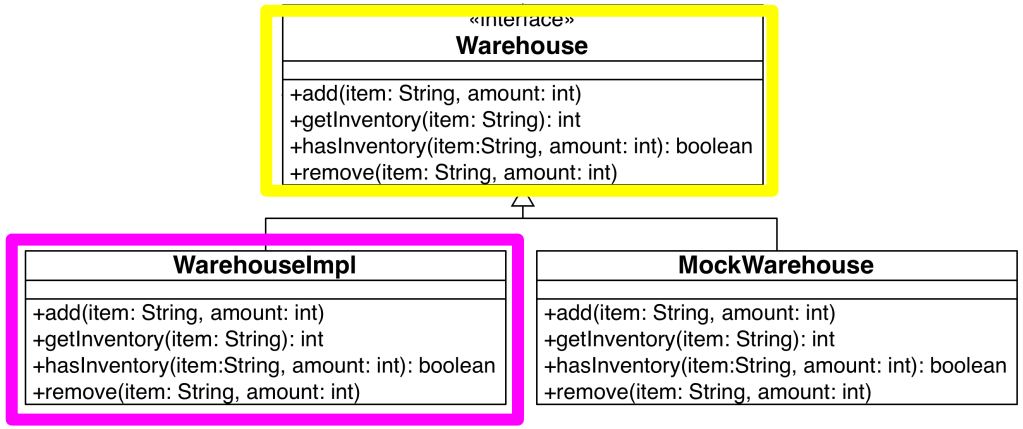
Here the injector is created. It allows us to specify bindings of implementations to interfaces. Uses a ProductionModule.

Two changes to the "Client"

Guice modules: configure the bindings

- Guice uses the concept of a Module to bind a subclass implementation to an interface
- Two modules often used: ProductionModule and TestModule
- Here is the Guice Module ProductionModule that binds WarehouseImpl to Warehouse:

```
public class ProductionModule implements Module {
    public void configure(Binder binder) {
        binder.bind(Warehouse.class).to(WarehouseImpl.class);
    }
}
```



Four steps to use Guice

1. Tell Guice where to inject using the @Inject annotation:
   - Constructor injection
   - Method injection
   - Field injection (with Java Annotation)

2. Create a Module to define the Binding

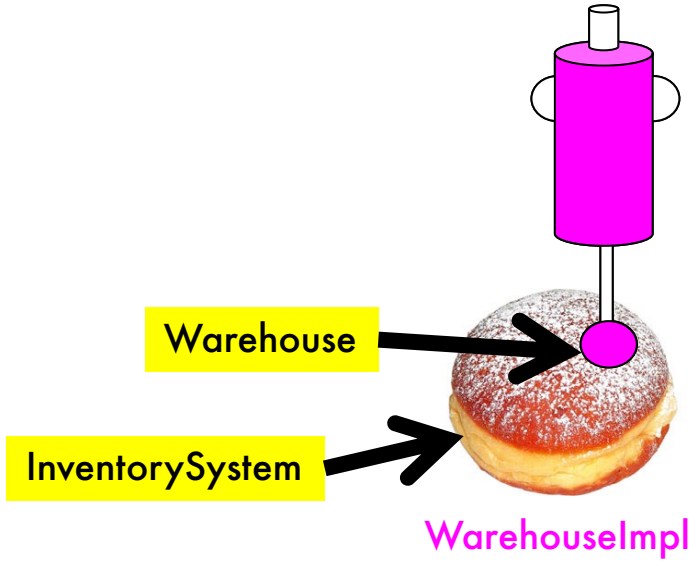
```
@Inject private Warehouse warehouse;

public class ProductionModule implements Module {
  public void configure(Binder binder) {
    binder.bind(Warehouse.class).to(WarehouseImpl.class);
  }
}
```

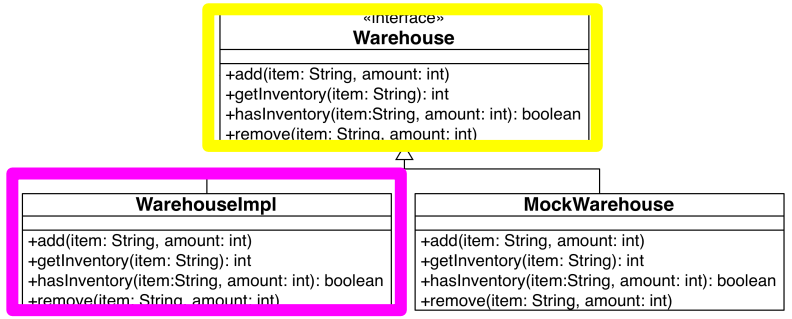
3. Instantiate an Injector and tell it which Module to use (i.e. provide a list of available bindings)

```
Injector injector = Guice.createInjector(new ProductionModule());
```

4. Instantiate an instance of the class needing the injection (in our case: InventorySystemDI)

```
InventorySystemDI inventorySystem = injector.getInstance(InventorySystemDI.class);
```





Guice and unit testing

- How does this help us with unit testing?
- Remember – instead of this hard-coded dependency:

```
public class InventorySystem {
    private Warehouse warehouse = new WarehouseImpl();
    …
}
```

we now have:

```
public class InventorySystemDI {
    @Inject private Warehouse warehouse;
    …
}
```

Injector filled with Warehouse

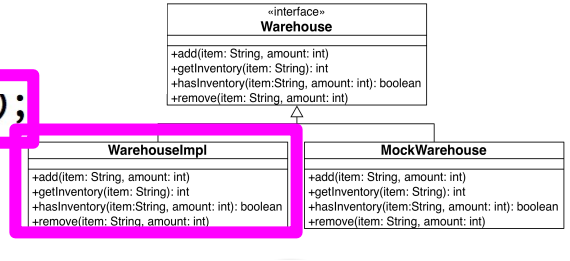
InventorySystem

- As a result we are able to write a unit test that binds warehouse to a mock object
- To do this we define another Guice Module called TestModule

Configuring the bindings

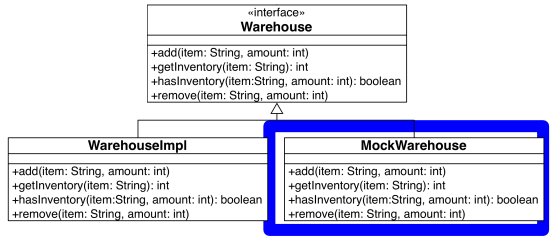
- The Guice module for the production code:

```
public class ProductionModule implements Module {
    public void configure(Binder binder) {
        binder.bind(Warehouse.class).to(WarehouseImpl.class);
    }
}
```



- The Guice module for the unit test:

```
public class TestModule implements Module {
    public void configure(Binder binder) {
        Warehouse warehouseMock = EasyMock.createMock(Warehouse.class);
        binder.bind(Warehouse.class).toInstance(warehouseMock);
    }
}
```



Testing inventory system with Guice & Easy mock

```
public class InventorySystemDITest {
    private static String TALISKER = "Talisker";
    private InventorySystemDI inventorySystem;
    private Injector injector;

    @Before public void setUp() throws Exception {
        injector = Guice.createInjector(new TestModule());
        inventorySystem = injector.getInstance(InventorySystemDI.class);
    }

    @Test public void addToWarehouse() {
        Warehouse warehouseMock = injector.getInstance(Warehou
        warehouseMock.add(TALISKER, 50);
        EasyMock.replay(warehouseMock);
        inventorySystem.addToWarehouse(TALISKER, 50);
        assertEquals(inventorySystem.getTotalStock(),
        EasyMock.verify(warehouseMock);
    }
}
```

We tell Guice to create an injector using TestModule, that binds Warehouse to a Warehouse Mock

We let Guice instantiate the InventorySystem and the Warehouse in it

We retrieve the mock Warehouse from the injector to use it in the test

Easy Mock Validation

Specified behavior in the test case

Summary

1. Simple unit tests allow to test and verify the functionality of a class, method, or a subsystem
2. Junit is a popular unit testing framework for Java, with various utility methods and annotations
3. The Mock object Pattern allows to also test the unit's behavior
4. We can achieve low coupling between the system and the tests with the Dependency Injection Pattern