Unit testing techniques and Role of Test doubles

Ritesh Mehrotra

What to test?

Everything!

Join at menti.com | use code 5578 7464

Ment

RM

Account

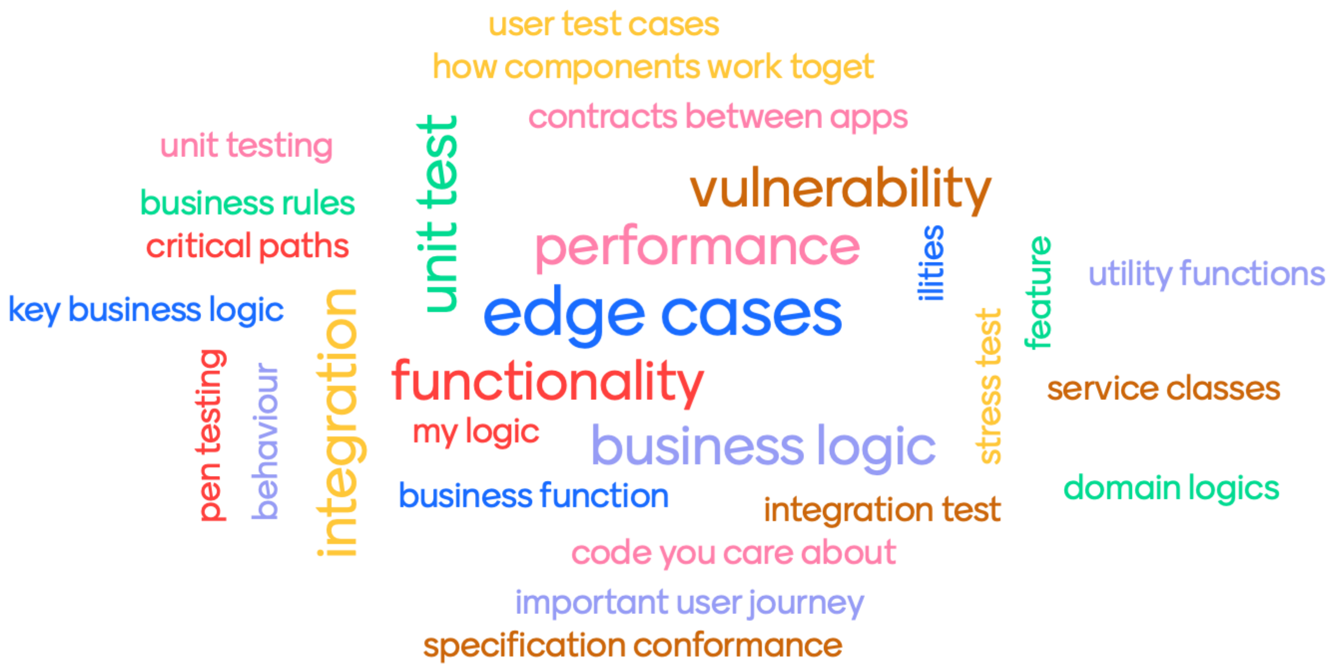
Content

Design

Settings

What to test?

36 responses



Help & Feedback

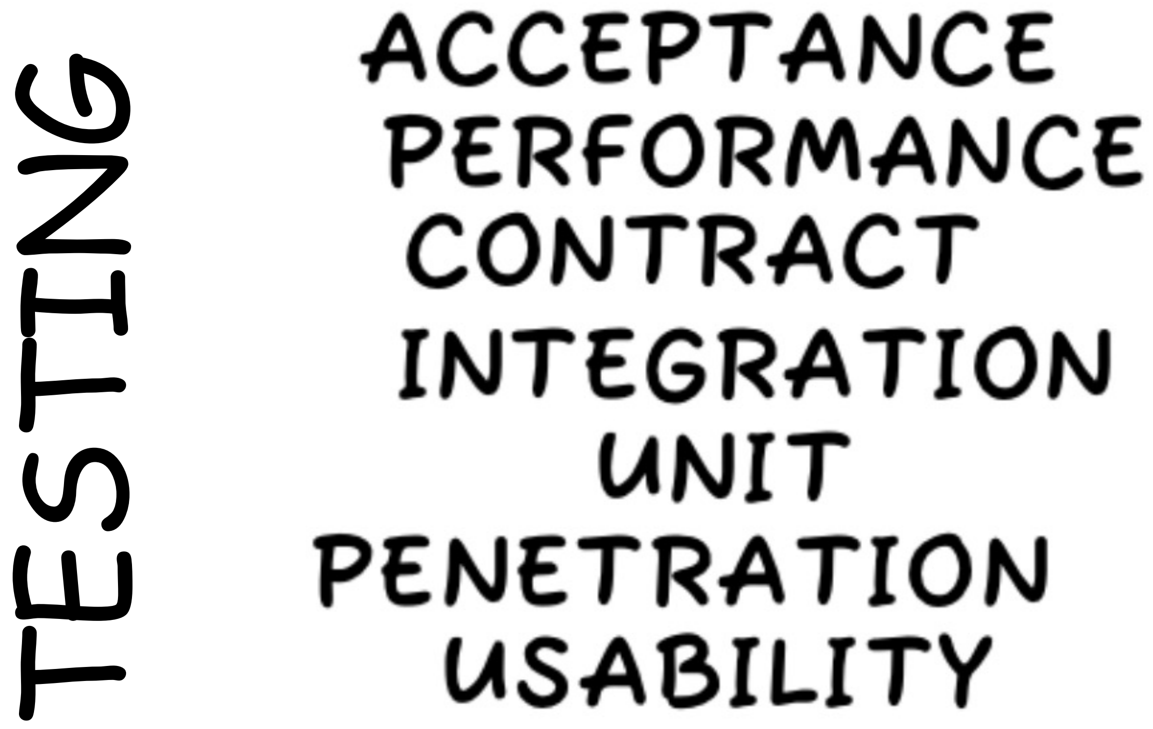
Consumer perspectives

- Does it work as expected?
- Is my data safe?
- Is it easy to use?
- Is it reliable?

Provider perspectives

- Does application work as expected?
- Is customer data safe?
- Is application easy to use?
- Is application reliable?
- Is application scalable?
- Is application maintainable?
- Is application reusable?

Types of tests



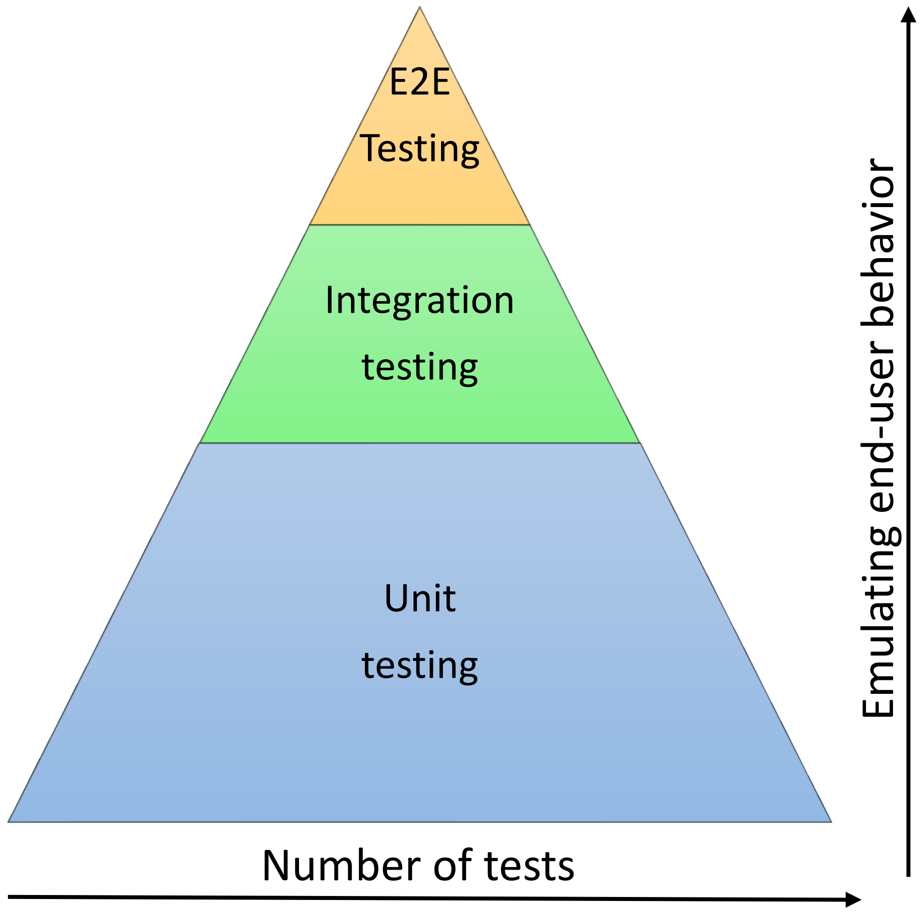
Today’s focus

- Unit testing patterns and anti-patterns
- Test doubles - Fakes, Stubs, Mocks, Dummies and Spies

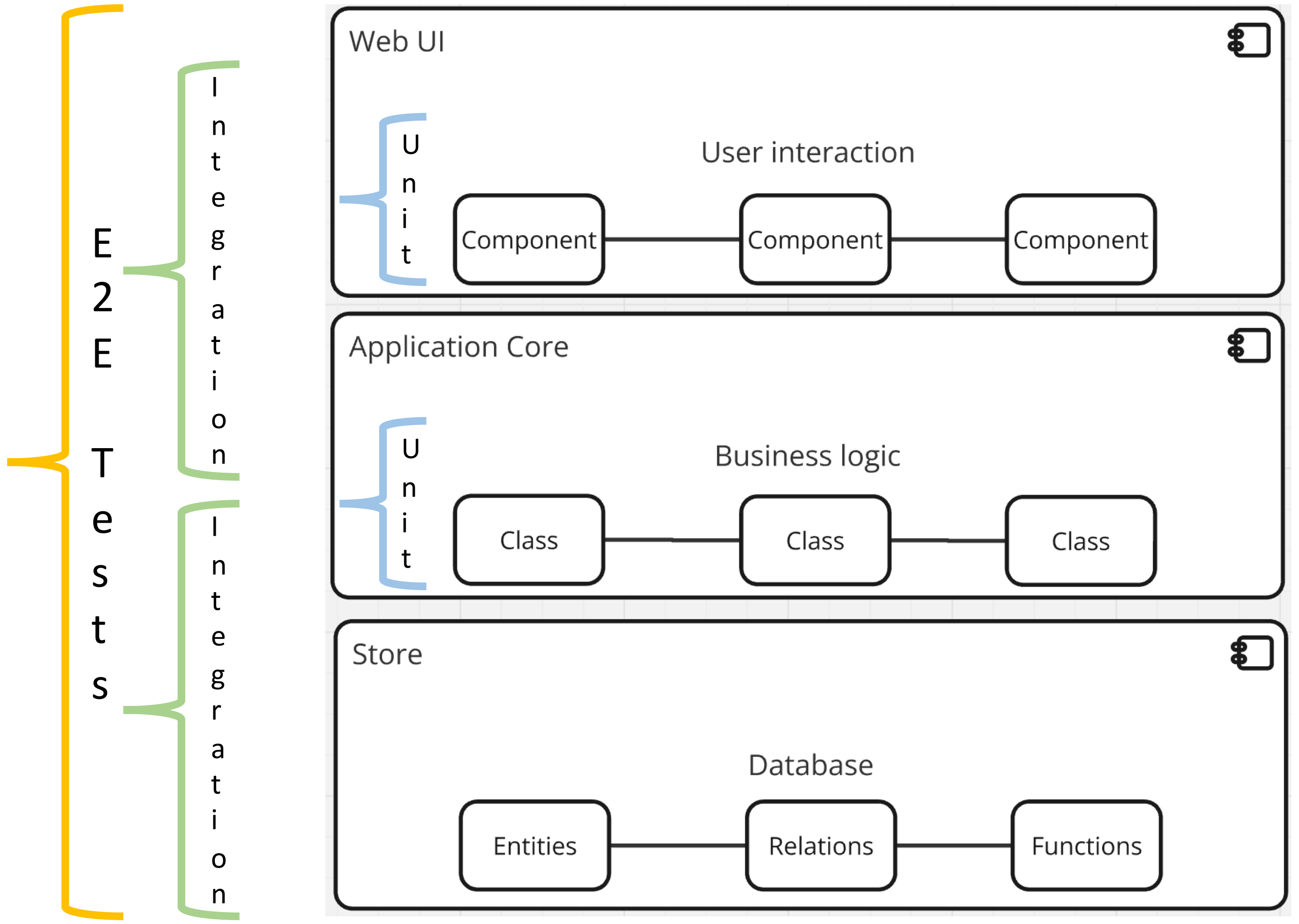
Approach to testing

- Validate the application use cases
  - Positive scenarios
  - Negative scenarios
  - Side effects
  - Exceptional flow
- Interaction between different components
- End to end flow of application

Test pyramid



Layered N-tier architecture



Unit Testing vs Integration Testing

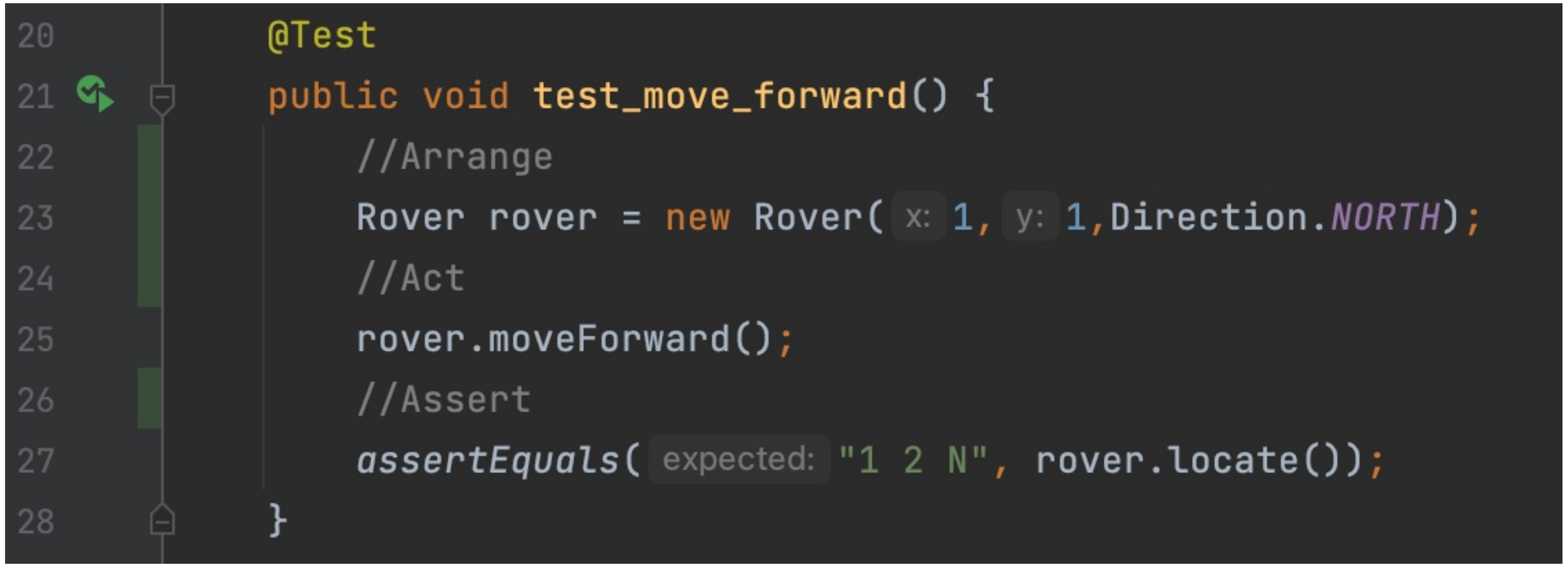
Characteristics	Unit Test	Integration Test
Interface	Works independently of real interface (File system, Database, API)	Depends on interface
Time	Quick to run	Time consuming operation
Reliability	Very reliable since meant to be isolated	Flaky at time depending on environment stability
Target	Tests behavior of the code	Tests behavior as well as interactions between objects
Environment	Environment independent	Environment dependent

Struggles with Unit testing

- Not valuable enough?
- Time consuming exercise
- Contributing factors
    - Code coupling makes maintainability and testability harder
    - Complex code leads to complex and flaky tests
    - Tests need to be updated every time code structure changes
- Unit testing for sake of code coverage?

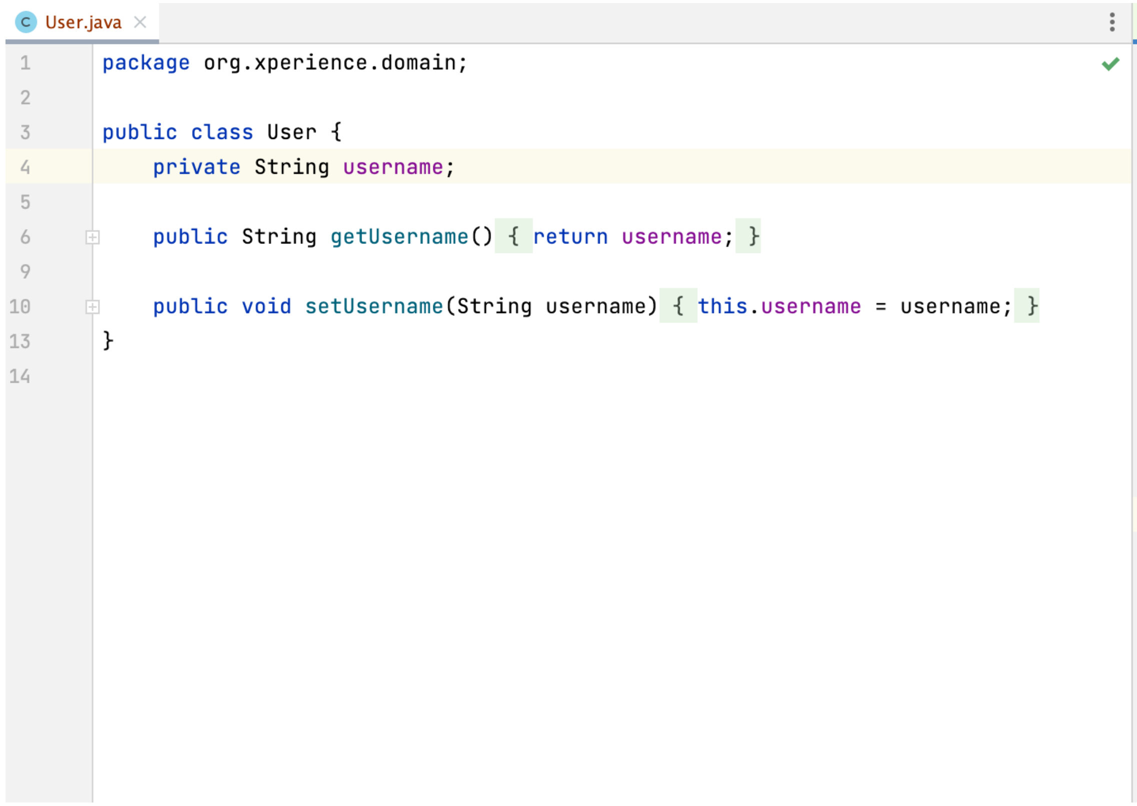
Structure of a test

- Arrange – Bring SUT and its dependencies to a desired state
- Act – Call methods on SUT and capture the return value (if any)
- Assert – Verify the outcome



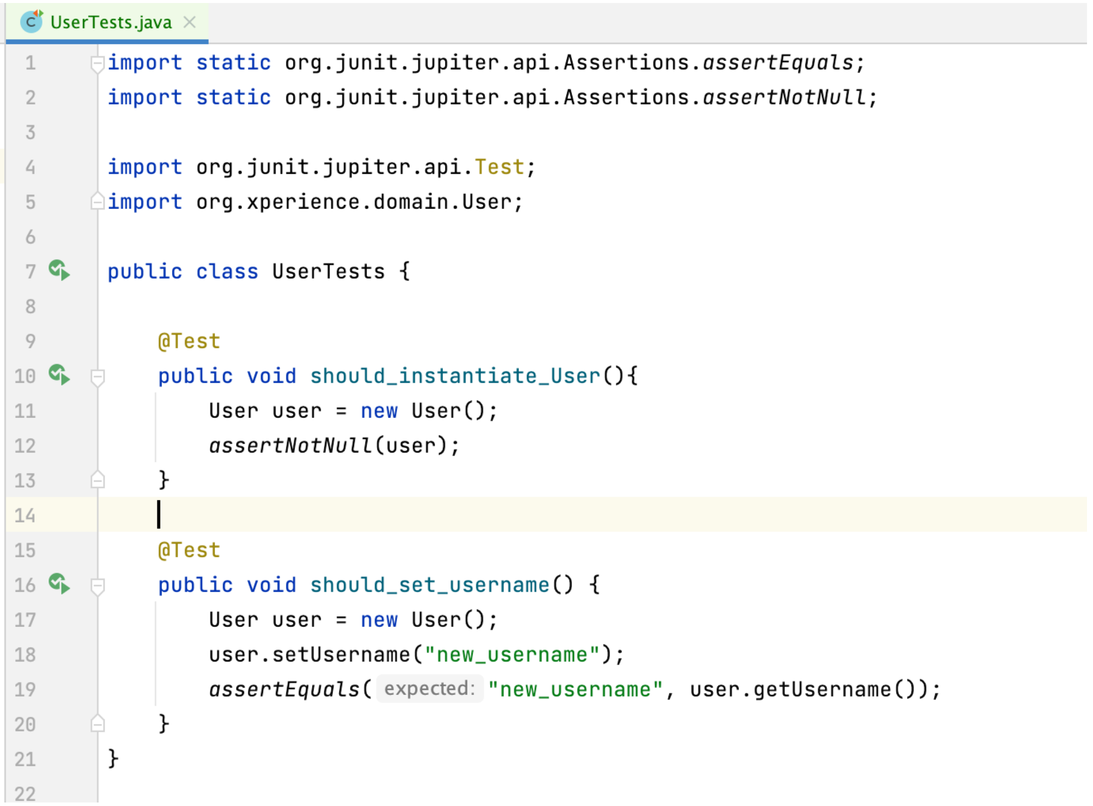
```
    @Test
    public void test_move_forward() {
        //Arrange
        Rover rover = new Rover( x: 1, y: 1,Direction.NORTH);
        //Act
        rover.moveForward();
        //Assert
        assertEquals( expected: "1 2 N", rover.locate());
    }
```

Trivial tests



User.java

```java
package org.xperience.domain;

public class User {
    private String username;

    public String getUsername() { return username; }

    public void setUsername(String username) { this.username = username; }
}
```



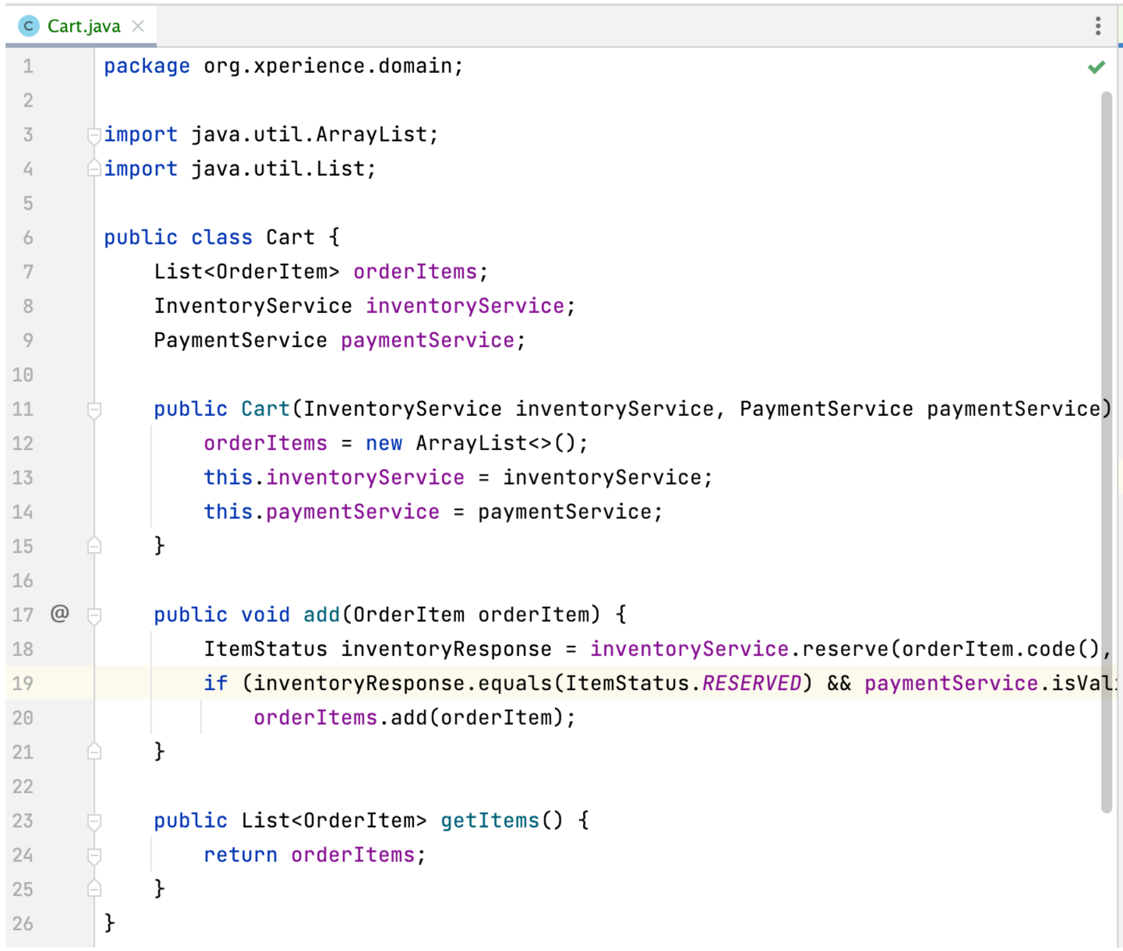
UserTests.java

```java
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;

import org.junit.jupiter.api.Test;
import org.xperience.domain.User;

public class UserTests {

    @Test
    public void should_instantiate_User(){
        User user = new User();
        assertNotNull(user);
    }

    @Test
    public void should_set_username() {
        User user = new User();
        user.setUsername("new_username");
        assertEquals( expected: "new_username", user.getUsername());
    }
}
```

Its not valuable if we our tests assert on

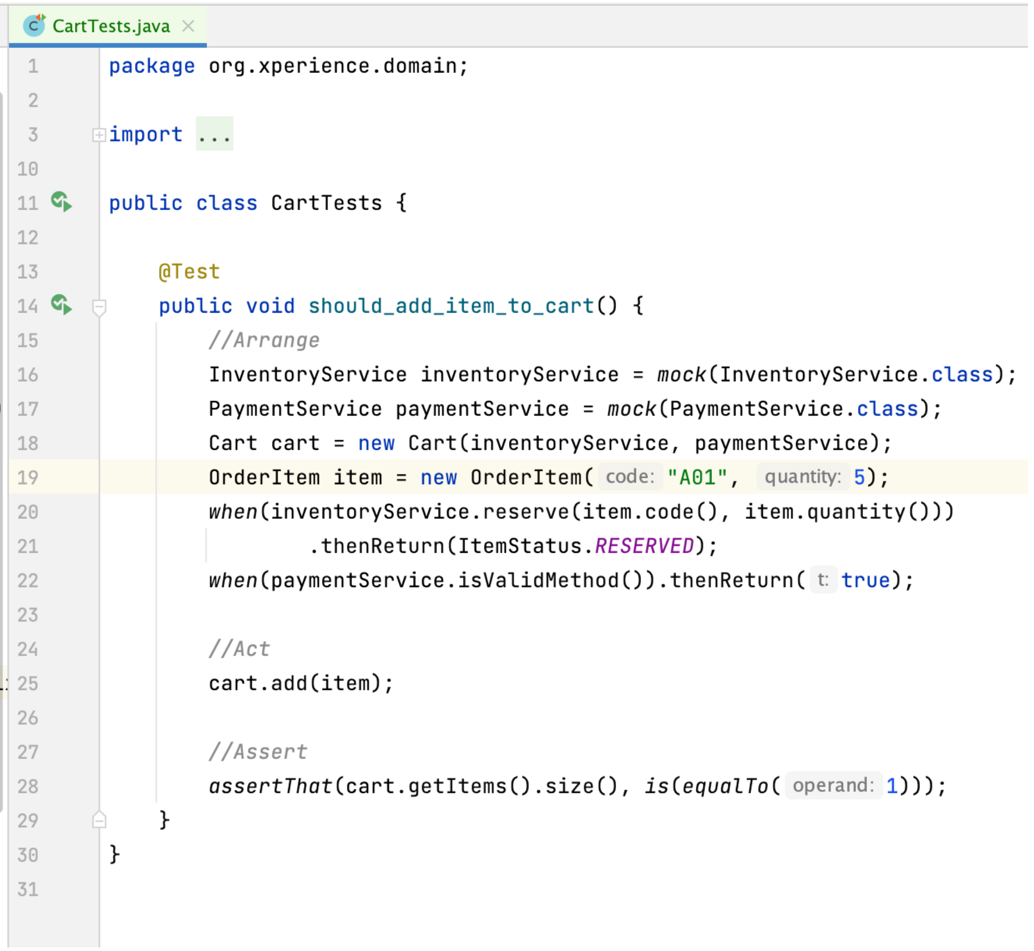
- Getters and Setters
- Object instantiation
- Composition of a class with its coordinating classes
- A mathematical formula like multiplication or square root

Complex test



Cart.java

```java
package org.xperience.domain;

import java.util.ArrayList;
import java.util.List;

public class Cart {
    List<OrderItem> orderItems;
    InventoryService inventoryService;
    PaymentService paymentService;

    public Cart(InventoryService inventoryService, PaymentService paymentService)
        orderItems = new ArrayList<>();
        this.inventoryService = inventoryService;
        this.paymentService = paymentService;
    }

    public void add(OrderItem orderItem) {
        ItemStatus inventoryResponse = inventoryService.reserve(orderItem.code(),
        if (inventoryResponse.equals(ItemStatus.RESERVED) && paymentService.isVal
            orderItems.add(orderItem);
    }

    public List<OrderItem> getItems() {
        return orderItems;
    }
}
```

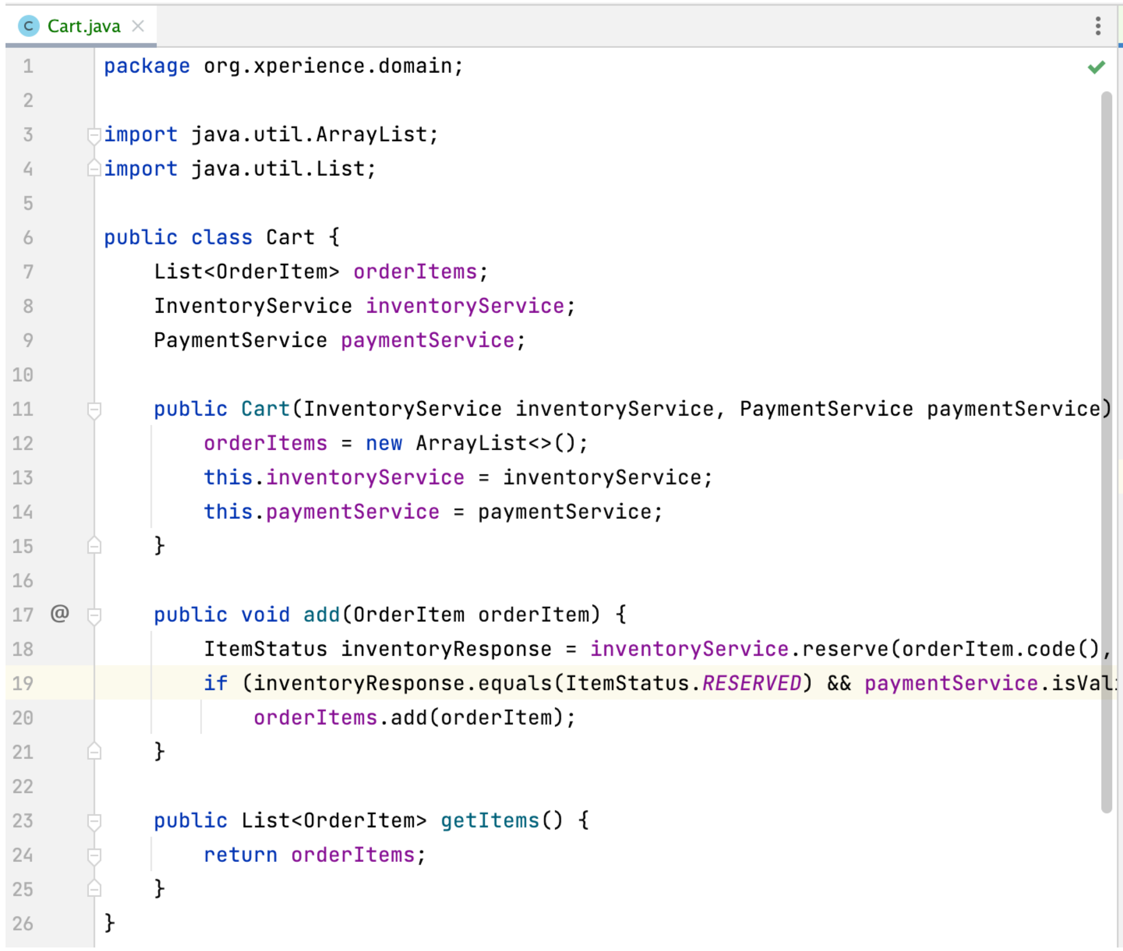


CartTests.java

```java
package org.xperience.domain;

import ...

public class CartTests {

    @Test
    public void should_add_item_to_cart() {
        //Arrange
        InventoryService inventoryService = mock(InventoryService.class);
        PaymentService paymentService = mock(PaymentService.class);
        Cart cart = new Cart(inventoryService, paymentService);
        OrderItem item = new OrderItem( code: "A01",  quantity: 5);
        when(inventoryService.reserve(item.code(), item.quantity()))
                .thenReturn(ItemStatus.RESERVED);
        when(paymentService.isValidMethod()).thenReturn( t: true);

        //Act
        cart.add(item);

        //Assert
        assertThat(cart.getItems().size(), is(equalTo( operand: 1)));
    }
}
```

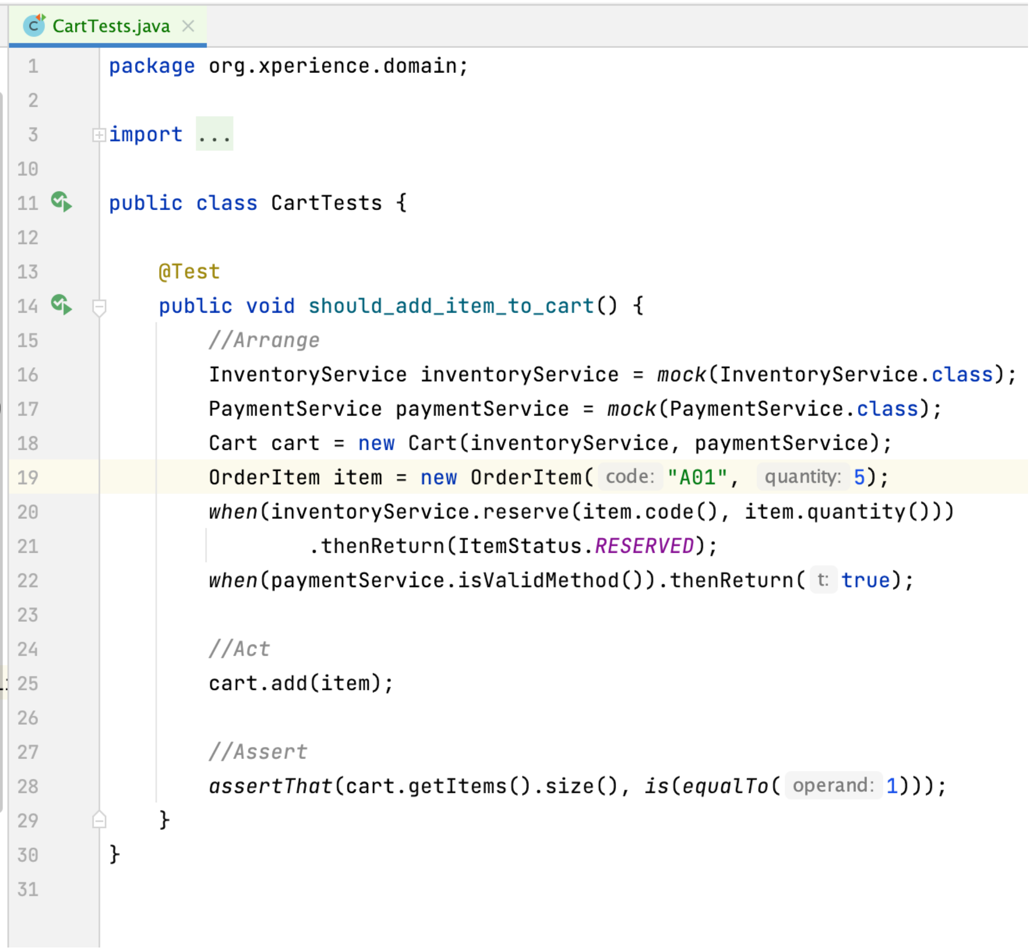
- Too large arrangement
- Indicative of an abstraction issue

Partial test



Cart.java

```java
package org.xperience.domain;

import java.util.ArrayList;
import java.util.List;

public class Cart {
    List<OrderItem> orderItems;
    InventoryService inventoryService;
    PaymentService paymentService;

    public Cart(InventoryService inventoryService, PaymentService paymentService)
        orderItems = new ArrayList<>();
        this.inventoryService = inventoryService;
        this.paymentService = paymentService;
    }

    public void add(OrderItem orderItem) {
        ItemStatus inventoryResponse = inventoryService.reserve(orderItem.code(),
        if (inventoryResponse.equals(ItemStatus.RESERVED) && paymentService.isVal
            orderItems.add(orderItem);
    }

    public List<OrderItem> getItems() {
        return orderItems;
    }
}
```

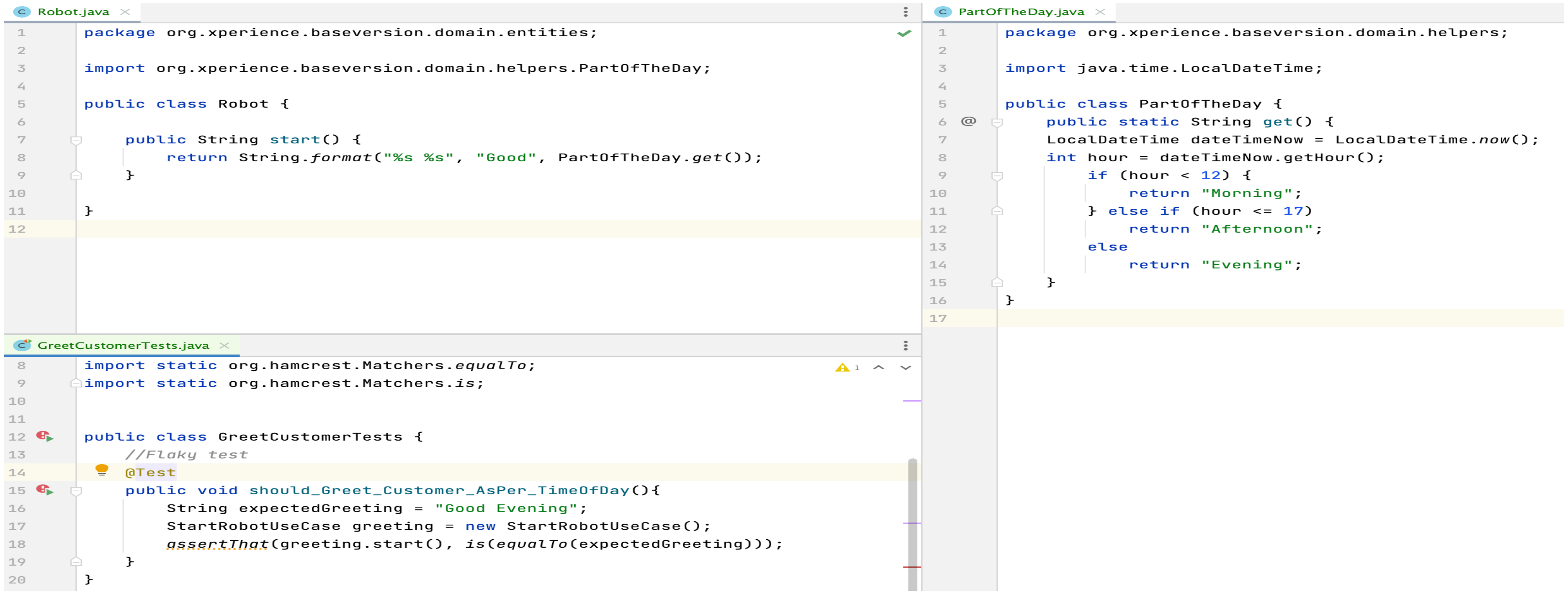


CartTests.java

```java
package org.xperience.domain;

import ...

public class CartTests {

    @Test
    public void should_add_item_to_cart() {
        //Arrange
        InventoryService inventoryService = mock(InventoryService.class);
        PaymentService paymentService = mock(PaymentService.class);
        Cart cart = new Cart(inventoryService, paymentService);
        OrderItem item = new OrderItem( code: "A01",  quantity: 5);
        when(inventoryService.reserve(item.code(), item.quantity()))
                .thenReturn(ItemStatus.RESERVED);
        when(paymentService.isValidMethod()).thenReturn( t: true);

        //Act
        cart.add(item);

        //Assert
        assertThat(cart.getItems().size(), is(equalTo( operand: 1)));
    }
}
```

Flaky test

- Tests that pass or fail inconsistently, even when the code hasn’t changed. These tests undermine confidence in the test suite.



Robot.java

```java
package org.xperience.baseversion.domain.entities;

import org.xperience.baseversion.domain.helpers.PartOfTheDay;

public class Robot {

    public String start() {
        return String.format("%s %s", "Good", PartOfTheDay.get());
    }

}
```

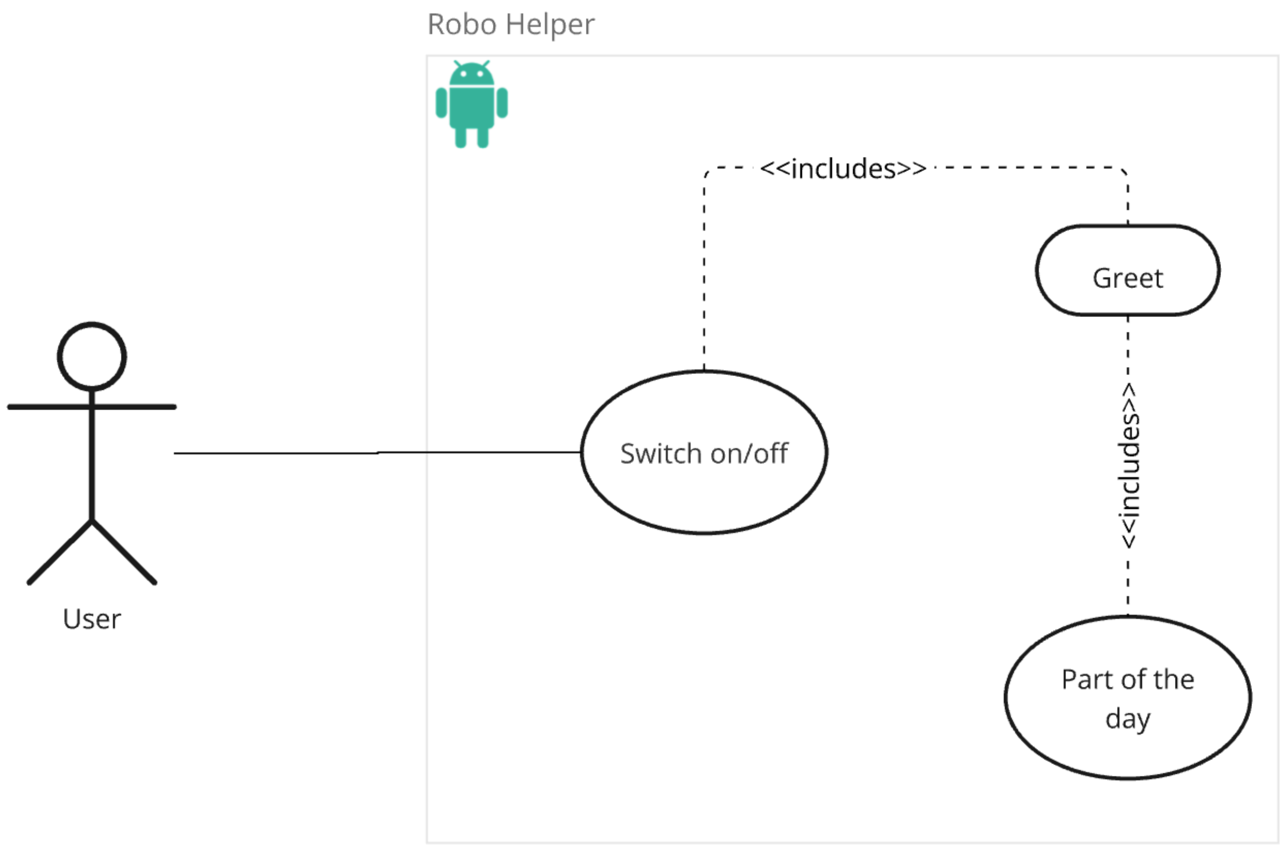
PartOfTheDay.java

```java
package org.xperience.baseversion.domain.helpers;

import java.time.LocalDateTime;

public class PartOfTheDay {
    public static String get() {
    LocalDateTime dateTimeNow = LocalDateTime.now();
    int hour = dateTimeNow.getHour();
        if (hour < 12) {
            return "Morning";
        } else if (hour <= 17)
            return "Afternoon";
        else
            return "Evening";
    }
}
```

GreetCustomerTests.java

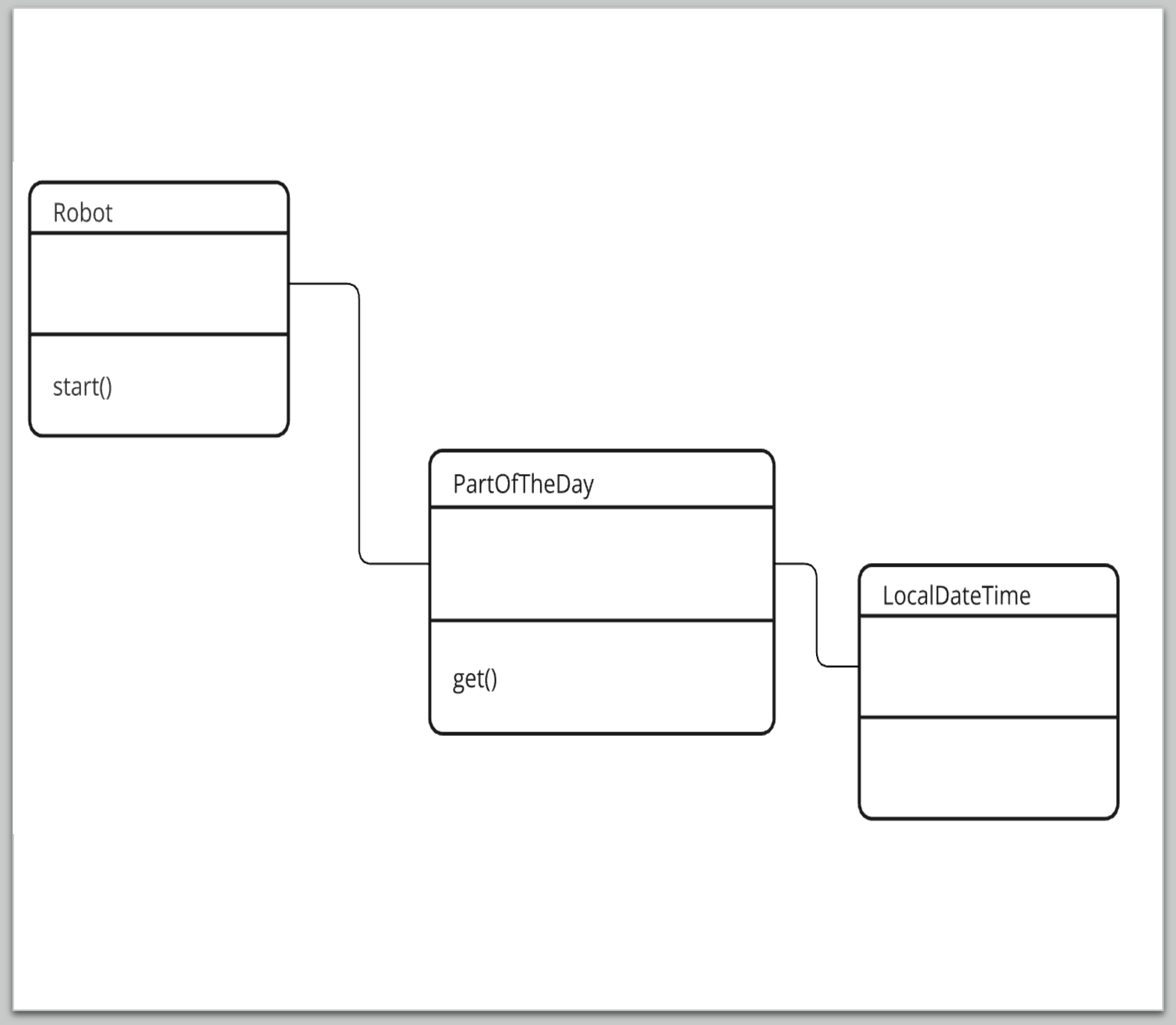
```java
import static org.hamcrest.Matchers.equalTo;
import static org.hamcrest.Matchers.is;


public class GreetCustomerTests {
    //Flaky test
    @Test
    public void should_Greet_Customer_AsPer_TimeOfDay(){
        String expectedGreeting = "Good Evening";
        StartRobotUseCase greeting = new StartRobotUseCase();
        assertThat(greeting.start(), is(equalTo(expectedGreeting)));
    }
}
```

Use Case – Robo Helper Start



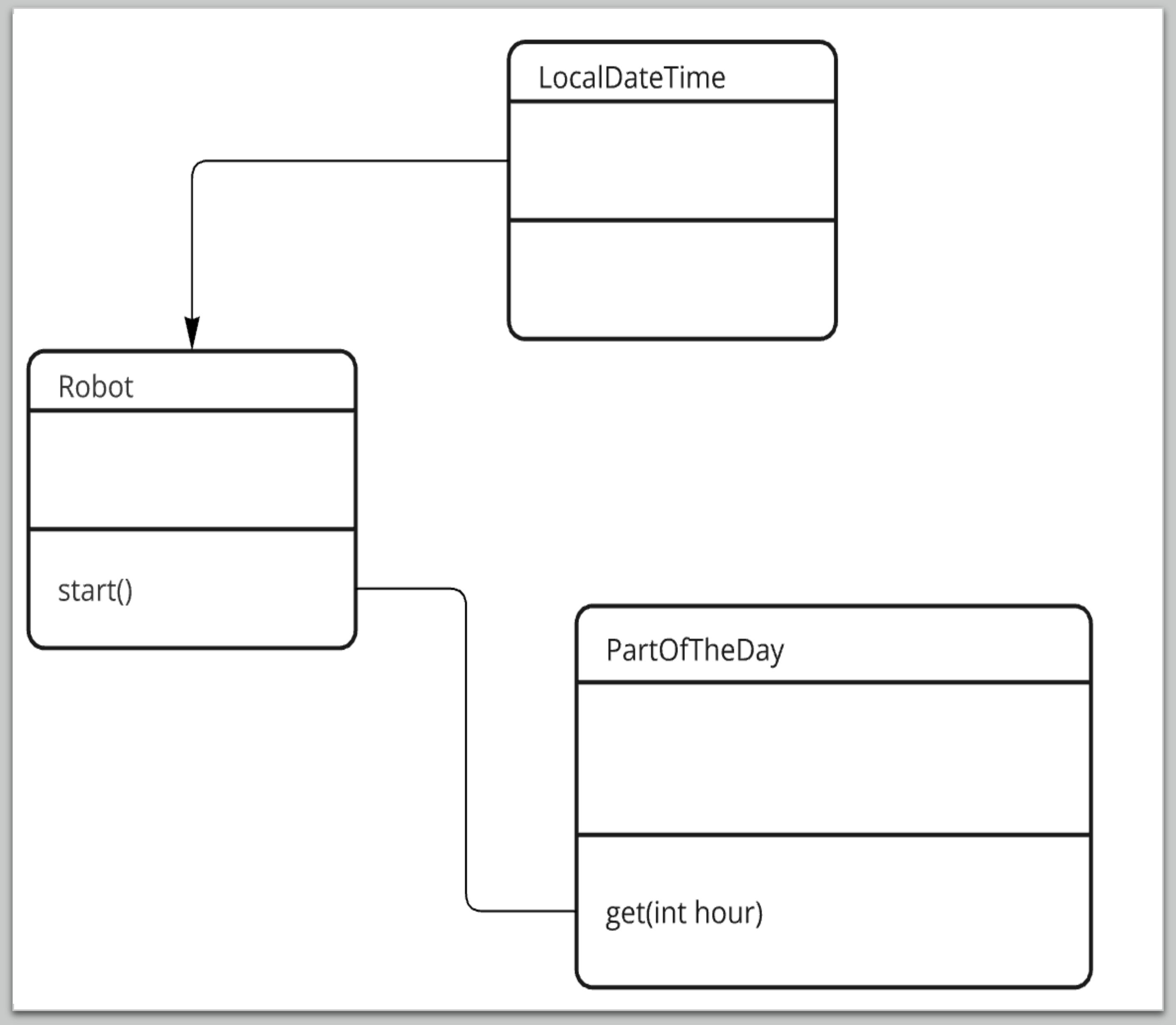
Challenges

- System time (Data source) tightly coupled within PartOfTheDay class
  - Can’t reuse this class with any other source of date/time
- It violates SRP – Fetches time and processes it
- Unit test cannot be deterministic



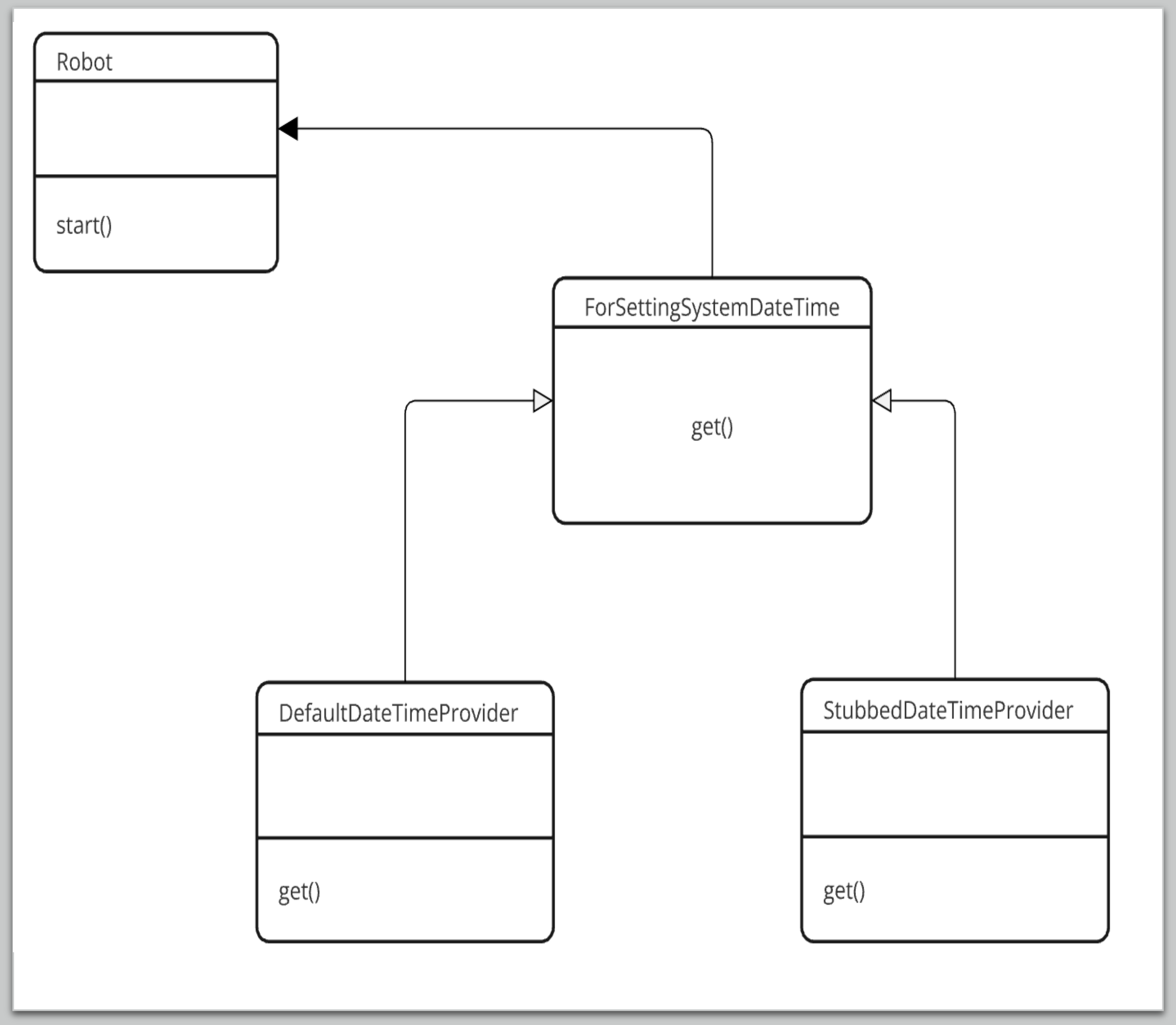
Consideration

Pass LocalDateTime as parameter to TimePeriod



Inversion of control

- Abstract concrete implementation to an interface
- Refactor to inject a test double implementation to Robot



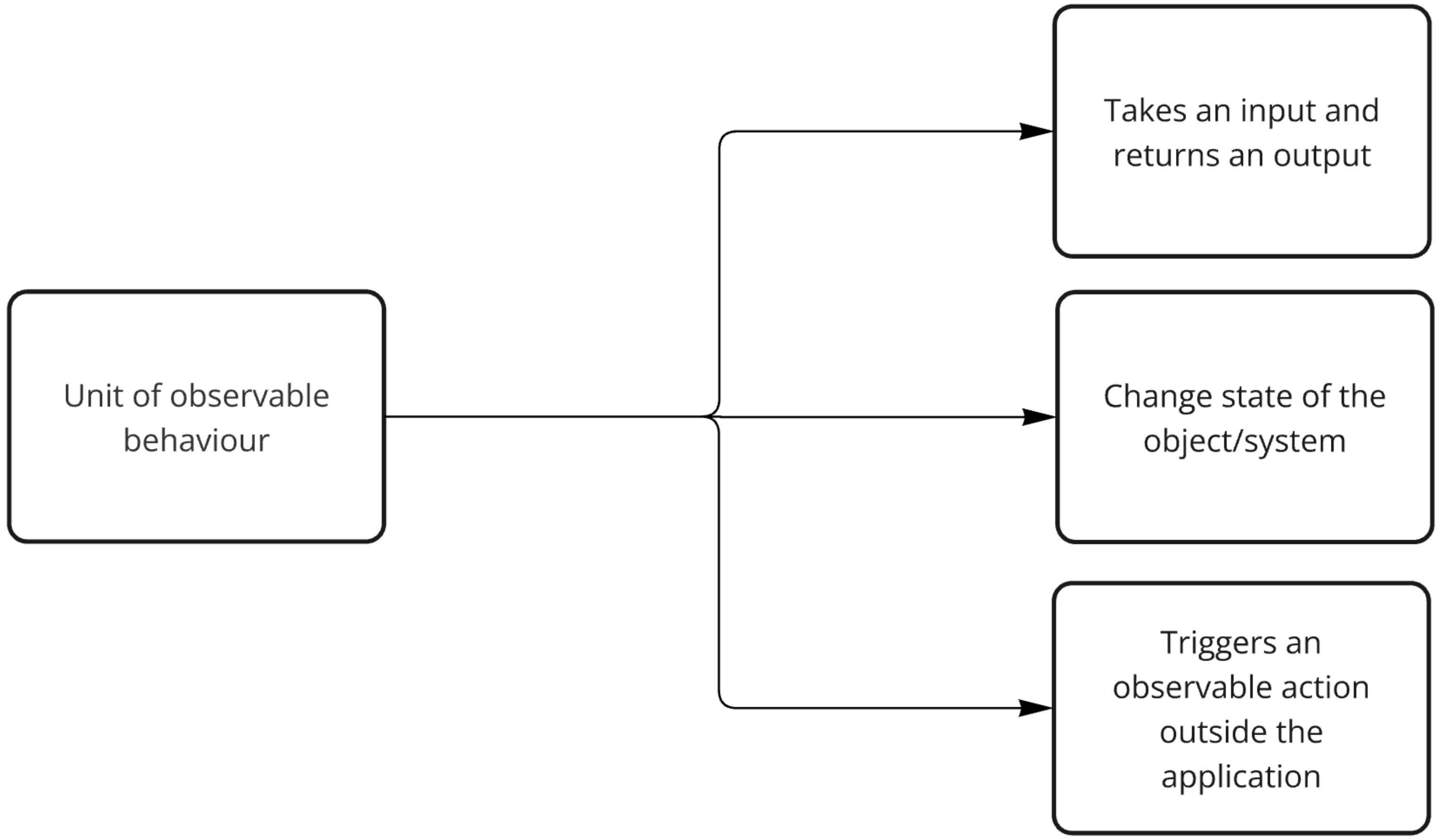
Good: Mocking tools lets you live with complexity

Bad: Mocking tools lets you live with complexity

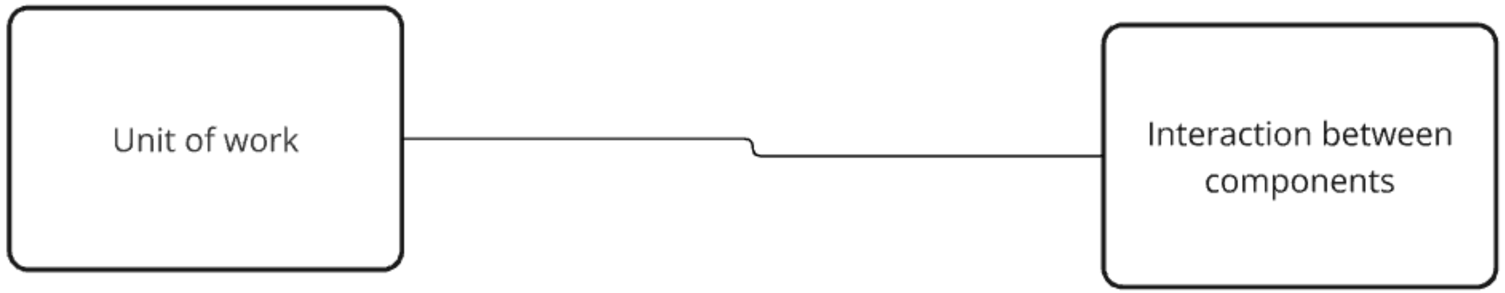
Good practices for unit testing

- Deterministic
- Automated
- Quick to run
- Should not fail when code’s internal structure changes
- Should fail when the behavior of code changes
- Cheap to read, write and change
- Tests should reduce (and not introduce) risk. Example: Private method made public for sake of testing
- Tests should be isolated and not dependent on each other

Unit of work(Behavior)


Unit of observable behaviour
Takes an input and returns an output
Change state of the object/system
Triggers an observable action outside the application

Unit of work (interaction)



Changing state or forwarding actions requires internal handling

- Application depends on another component for inputs (indirect input)
- Application produces certain outputs that cannot be tested (indirect outputs)

Test doubles

Types of test doubles

Stubs

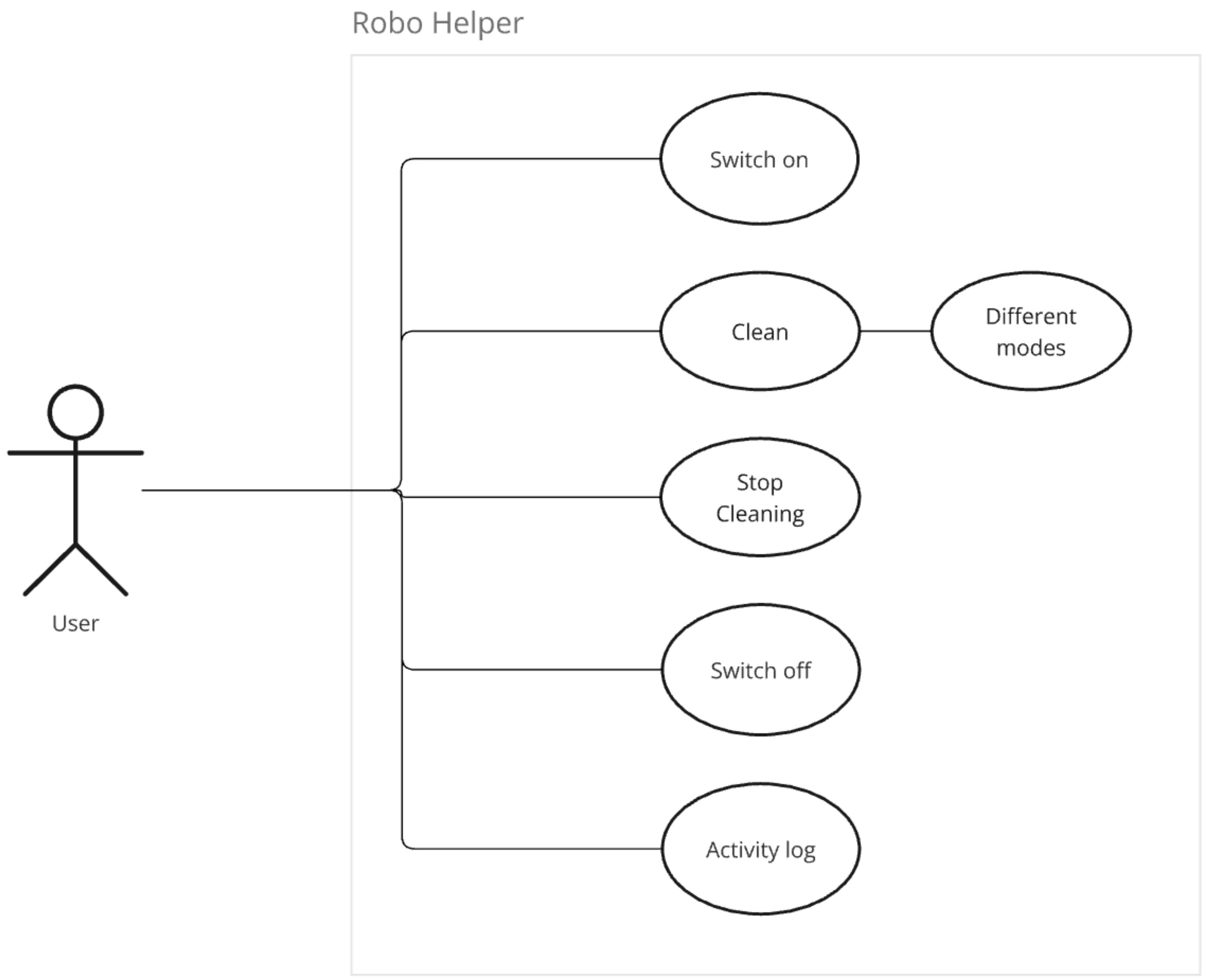
Dummy

Fakes

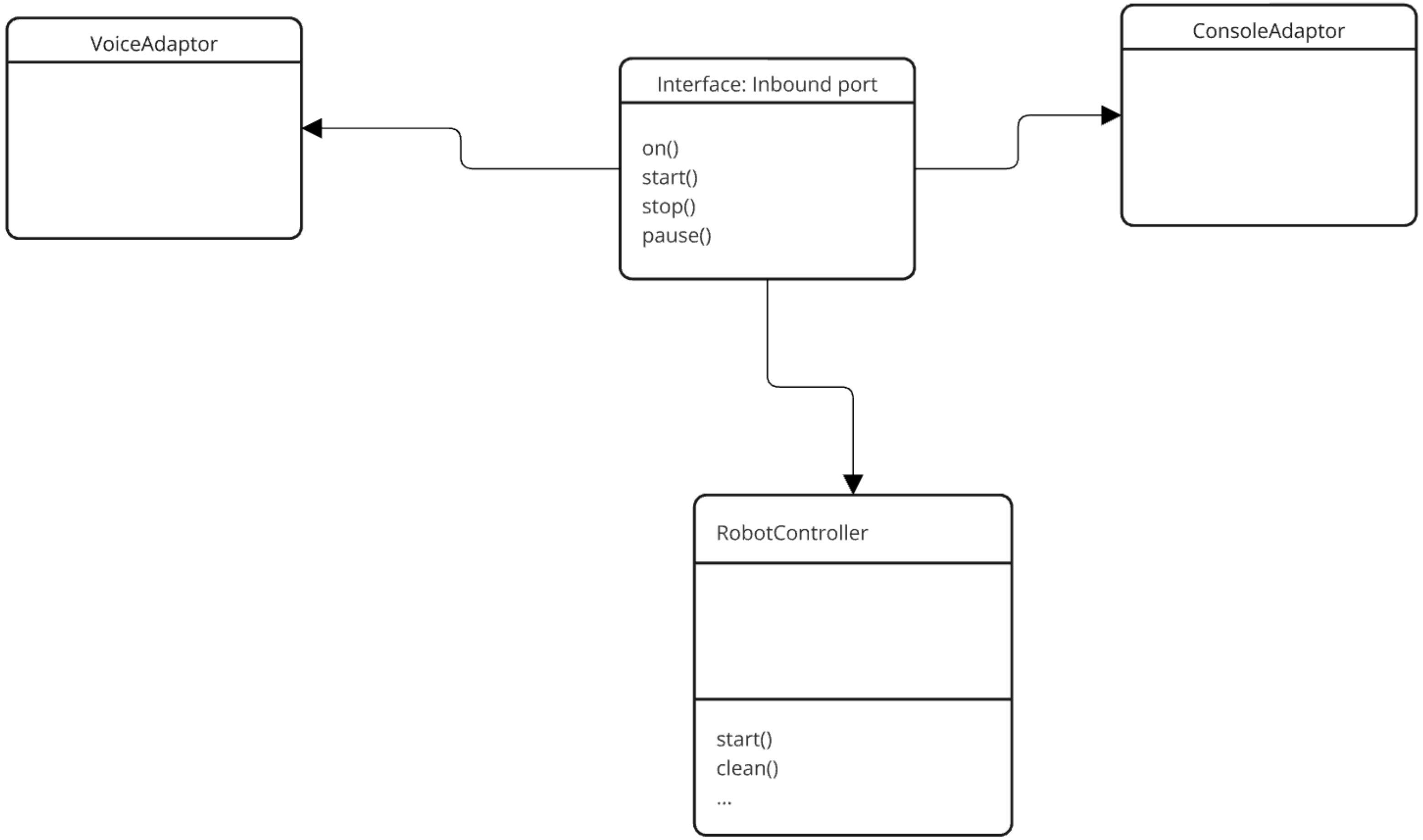
Mock

Spies

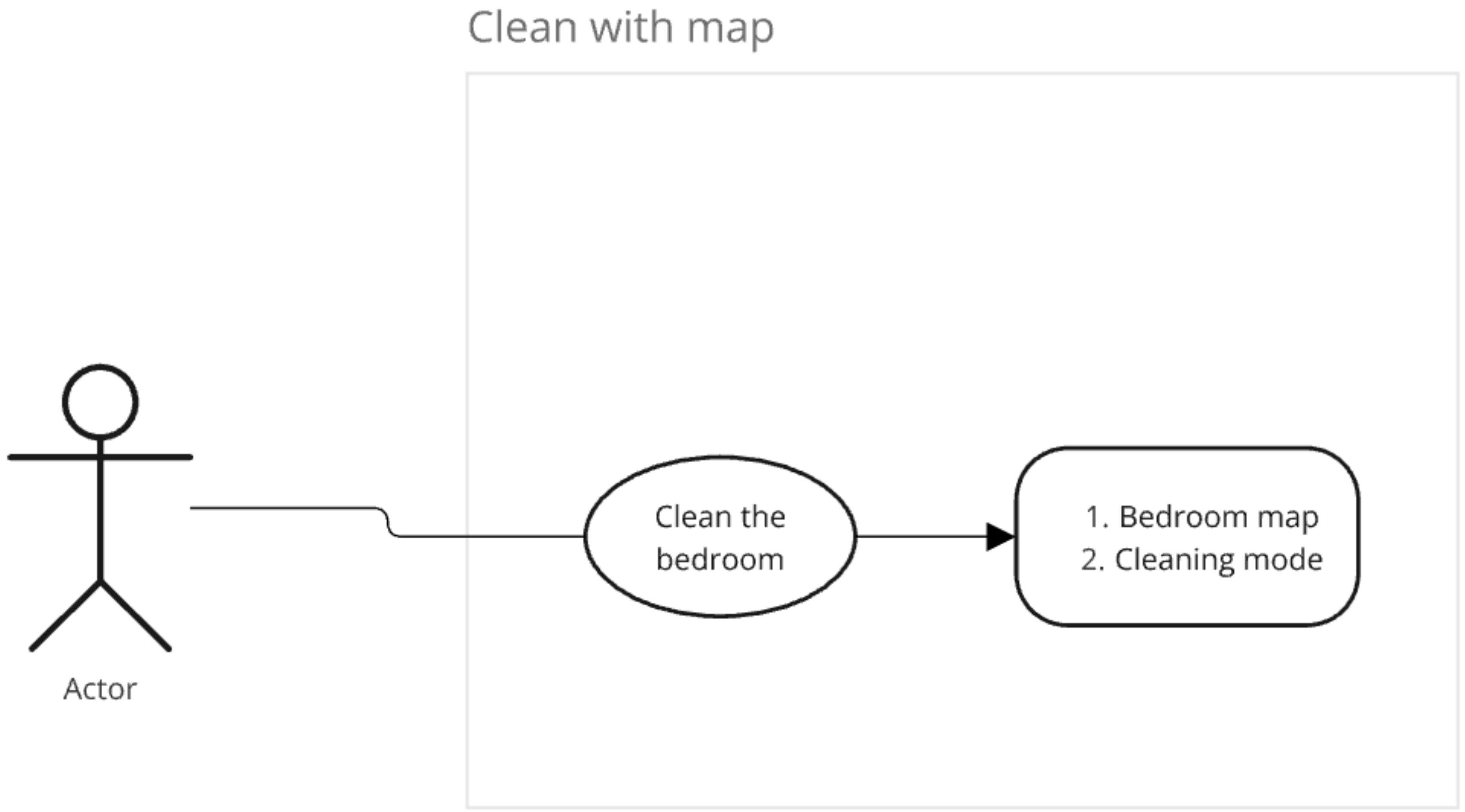
Use cases


Robo Helper
User
Switch on
Clean
Different modes
Stop Cleaning
Switch off
Activity log

Interaction with robot



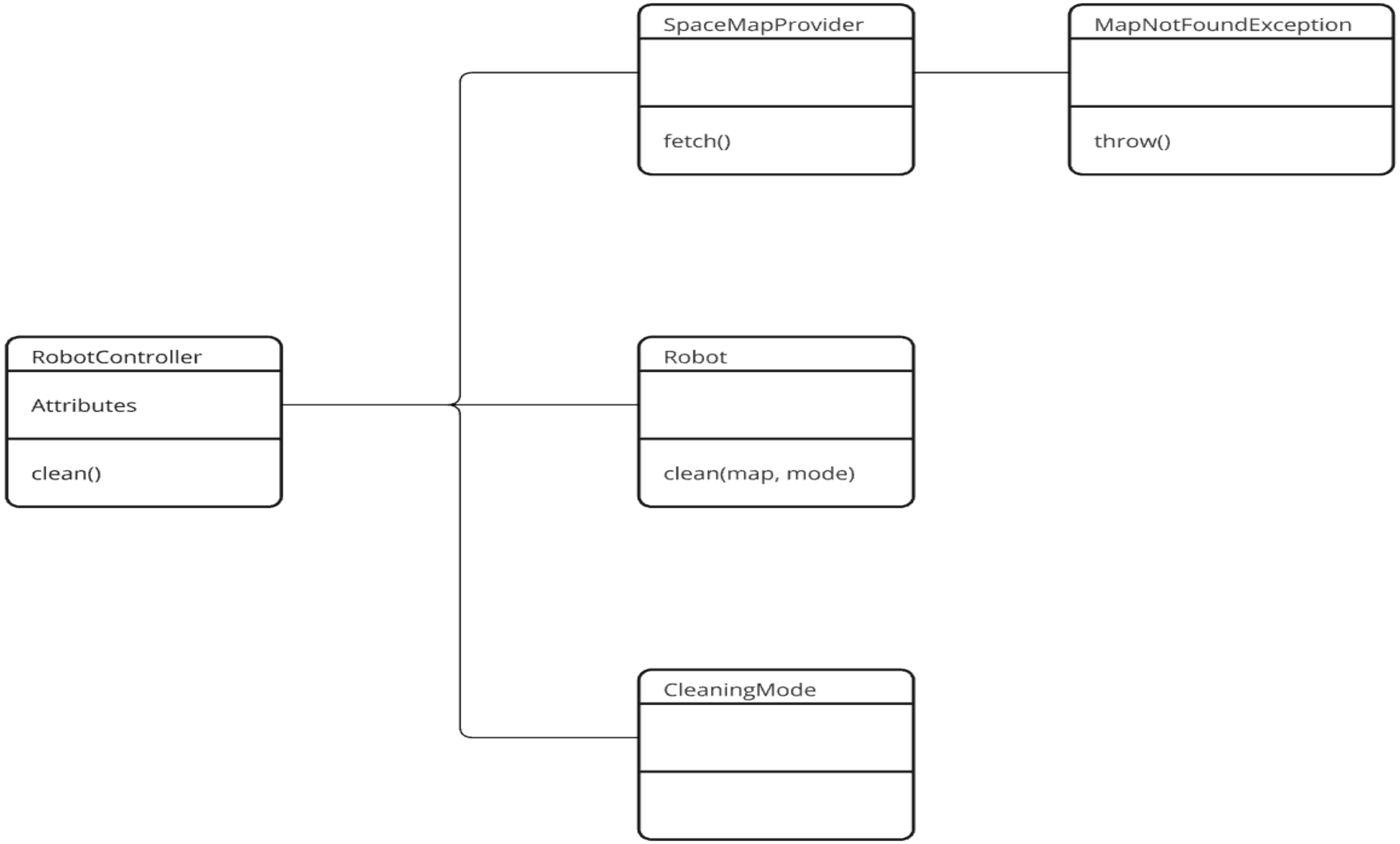
Use case – Clean with a given map



Scenarios – Cleaning pre-requisites

- Use case: Clean with provided map
  - Given a map is provided
    - Action: Start cleaning
    - Verification: Robot status should be cleaning
- Use case: Throw exception when map is not found
  - Given a map is not found
  - Action: Start cleaning
  - Verification: Exception thrown that Map is not found

Code Structure



Stub

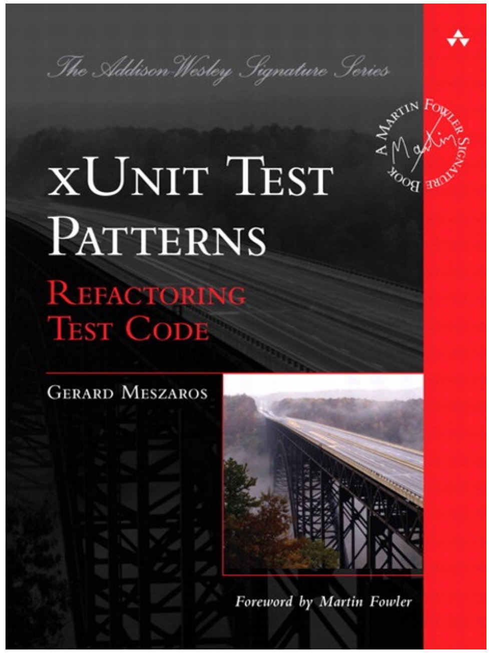
A filler object to satisfy behavioral needs of code through canned responses. Used as an alternative to expensive integration testing for testing dependency responses

Benefits of Stubs

- **Isolation**: The SUT is isolated from external dependencies, making tests faster and more reliable.
- **Controlled Data**: You can control the output of the stub to test various conditions without involving the real dependency.
- **Simplicity**: Stubs are simple to implement and provide a lightweight way to handle external dependencies in tests.

Types of stubs

- Responder- Returns a canned response
- Saboteur – Returns a canned exception


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

Scenarios – Cleaning pre-requisites

- Use case: Clean with provided map
  - Given a map is provided
    - Action: Start cleaning
    - Verification: <u>Robot</u> status should be cleaning
- Use case: Throw exception when map is not found
  - Given a map is not found
  - Action: Start cleaning
  - Verification: Exception thrown that Map is not found
- Challenge: How to deal with CleaningMode?
  - Not required as a pre-requisite testing, but expected in instantiation

Dummy

A filler object to satisfy structural needs of code under test. Used as a test helper and doesn’t impact the behavior of code under test.

Benefits of Dummy

- **Simplifies** test setup
- Keeps the test **focused** on the relevant behavior
- **Avoids unnecessary** logic or operations
- **Speeds up** test execution
- **Reduces dependencies**, making tests more isolated
- **Improves readability** of test code

Dummy is not same as Null object

- A Dummy object is not used by the SUT, so its behaviour is irrelevant. By contrast, a Null Object is used by the SUT
  - The null object is designed to do nothing.
  - Though may direct the logical flow

Scenarios – Cleaning with different modes

- Use case: Default cleaning mode
    - Given the cleaning mode is default and map is provided
        - Action: Start cleaning
        - Verification: Robot should <u>clean once</u> for the given map
- Use case: Deep cleaning mode
    - Given the cleaning mode is deep cleaning and map is provided
        - Action: Start cleaning
        - Verification: <u>Robot</u> should <u>clean twice</u> for the given map
- Challenge: How do I know if Robot cleaned once or twice? And for what map?

Spy

Spy is a stub that also records the interaction with caller objects

Benefits of Spy

- **Verifies** method calls and interactions.
- Tracks internal **method invocations** without breaking **encapsulation**
- Allows for **non-intrusive** observation of object behavior
- Facilitates partial mocking for specific methods
- Suitable for **interaction-based** testing
- Useful to test **indirect outputs**

Mock

Mock is a spy that verifies the interactions with caller object and fails if it doesn’t meet expectations. Unlike Spies, Mock doesn’t call the real implementation

Mock vs Spy

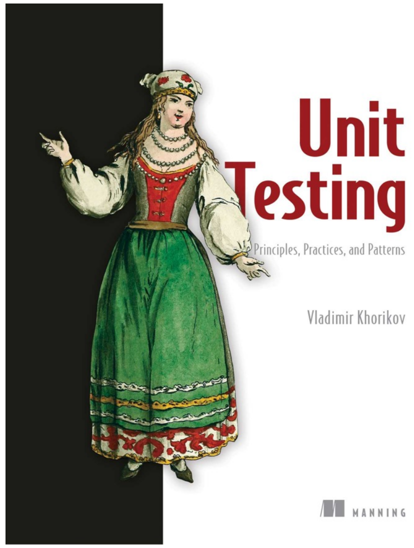
Feature	**Mock**	**Spy**
Definition	A mock is a fully simulated object that replaces the real object in a test. It doesn't call the real methods of the object.	A spy is a partial mock that wraps around a real object, allowing real methods to be called, but you can still verify interactions.
Purpose	Simulate behaviour of a dependency and verify specific method calls without affecting the actual implementation.	Track the behavior of a real object and verify interactions, while still calling the real methods.
Verification	Verification is primarily focused on method calls (how many times, with what arguments, etc.)	Records the interactions but does not verify. Provides a way to make assertions
Behaviour Simulation	Requires manual simulation of behavior using stubbing, defining how methods should act.	Can rely on the real behavior of the object, with options to override specific methods.
Complexity	Slightly simpler to implement since real methods aren't executed. Behavior is fully controlled by the test setup.	A bit more complex since it involves both executing real methods and verifying interactions.

Fake

A Fake object has a working implementation and behavior. Though meant for testing purpose and not suitable for production. For instance, using in-memory database instead of connecting to real database instance for operations

Benefits of using fake

- Closer to real implementation
- Not tied to structure of the implementation
- Helps focuses on behaviour instead of structure of program



Summary

- Unit tests help
  - Build developer confidence in code
  - Makes future changes easier
  - Helps to improve software quality
  - Helps to understand the system behaviour
  - Strengthens deployability of the code
- Use Test Doubles Wisely
  - Fakes: Simulate realistic behavior (e.g., in-memory databases).
  - Stubs: Provide fixed responses for isolation.
  - Mocks: Verify interactions and method calls.
  - Spies: Record method call details (e.g., call counts).
  - Dummy: As a test helper

Q & A

Let’s connect

- Linked In: https://www.linkedin.com/in/ritesh-mehrotra/
- Subscribe to my blog: https://neatstack.substack.com