Unit testing techniques and Role of Test doubles

Ritesh Mehrotra

TECHTALKS

About Techtalks

To build a community for collaborative and continuous learning

- Technology trends
- Tools and techniques
- Software engineering practices
- Ways of working
- Tech careers

TECHTALKS

About me

- Technical agility coach with Singapore Airlines
- Worked in Investment banking, Fintech and Software services
- Talk about Software craftsmanship, quality and agility
- Trainer for Agile Product and Delivery management (APM), Product Ownership(APO), Facilitation(ATF) and Foundation (ICP) with ICAgile
- Co-founded TechTalks in 2016

TECHTALKS

Agenda

- Foundation of Unit testing
  - Testing paradigm
  - Unit of work
- Organizing a unit test
- Testing with dependencies
- Test doubles - Fakes, Stubs, Mocks, Dummies and Spies
- Examples

Unit testing foundation

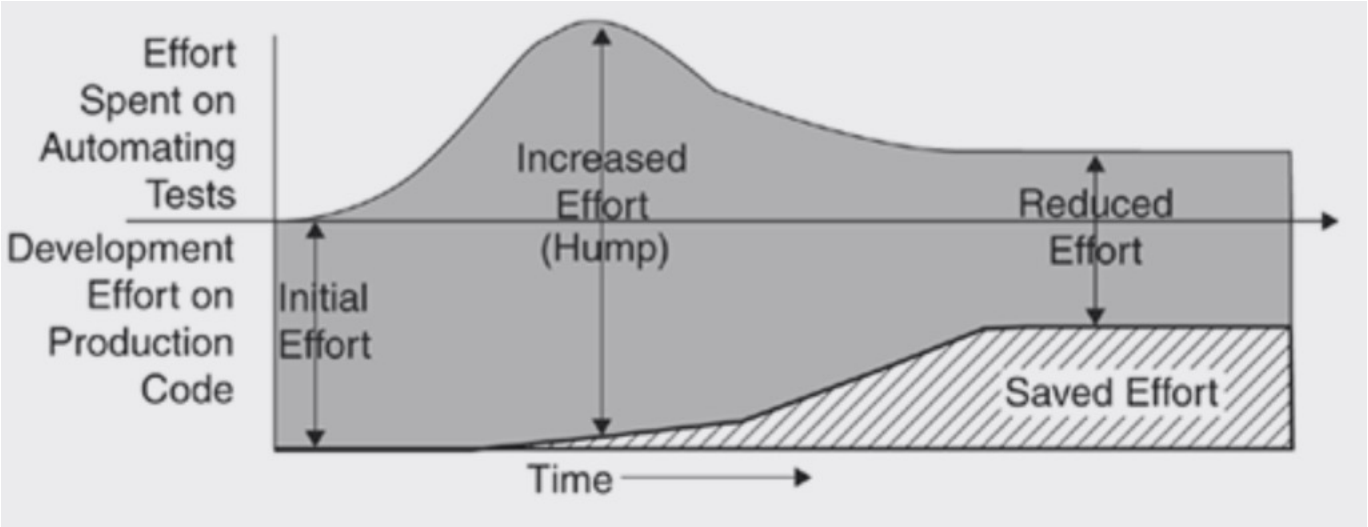
What is a unit test?

Validates the expectations with the code

Why unit tests?

- Builds developer confidence in code
- Makes future changes easier
- Helps to improve software quality.
- Helps to understand the system behaviour.
- Strengthens deployability of the code

Cost of automated testing



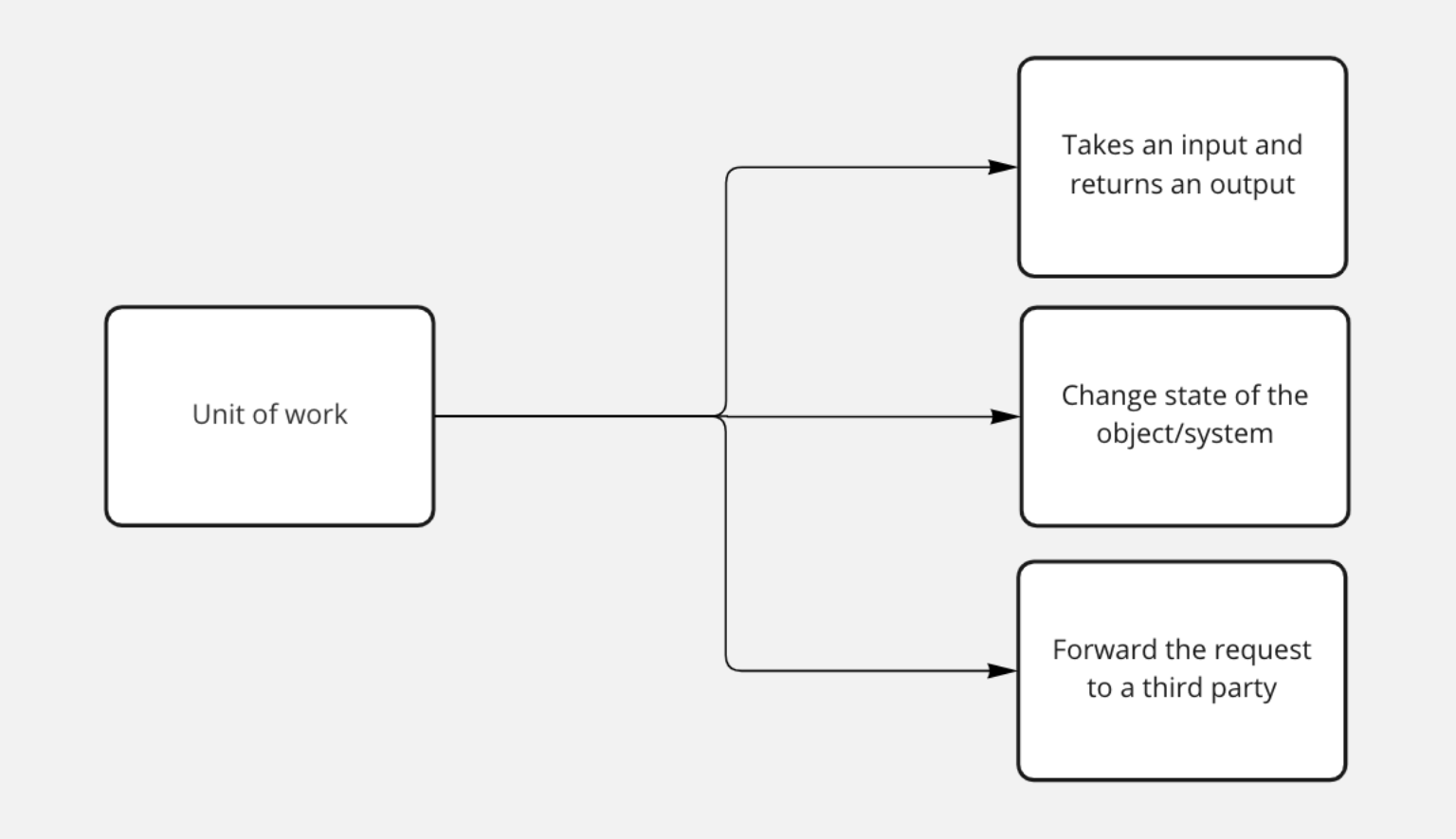
Reference: xUnit Test patterns – Gerard Meszaros

TECHTALKS

How big is a unit?

- Instantiating an object?
- Setting field values?
- Or A CRUD operation?

Unit of work



TECHTALKS

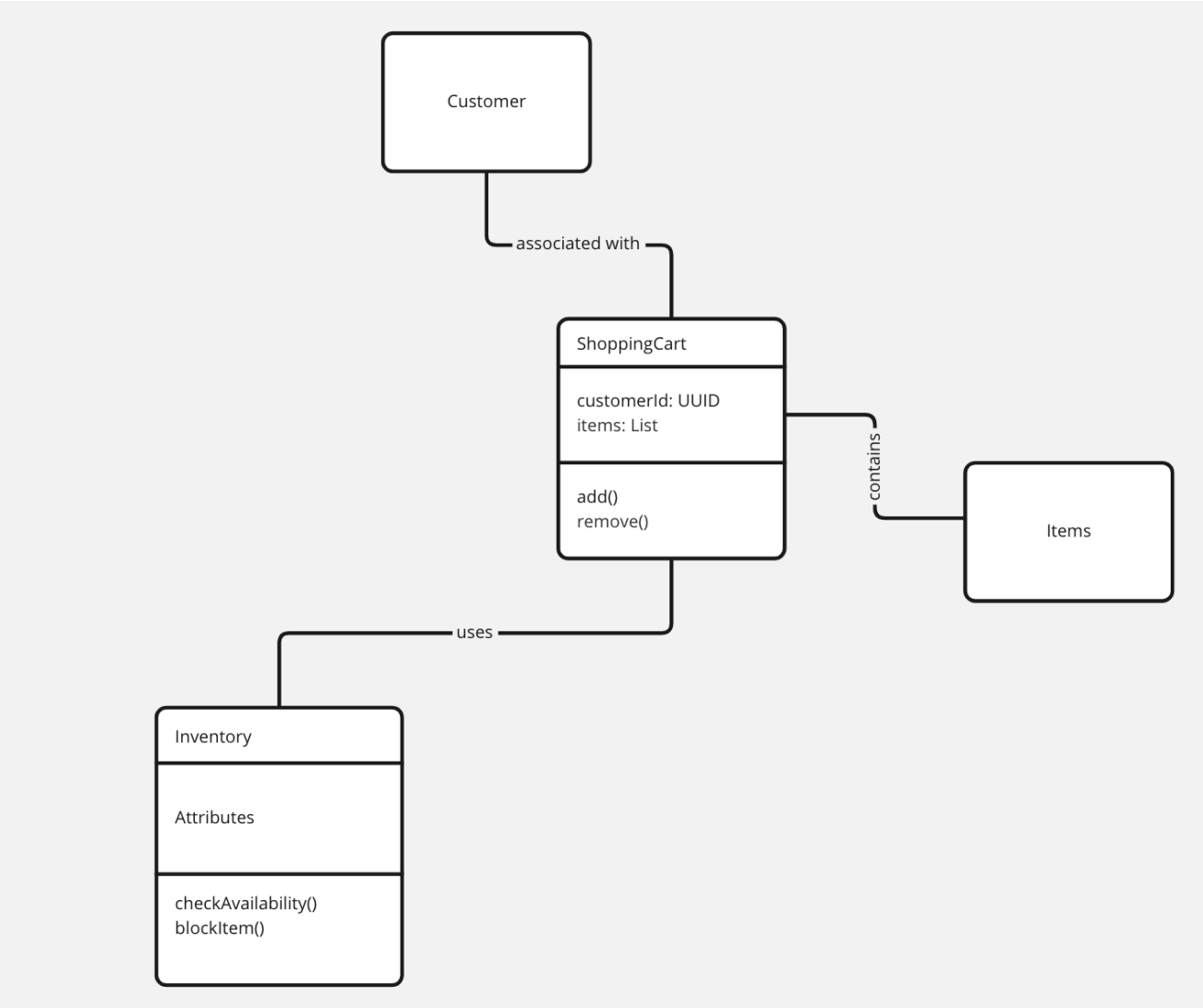
Terminology

- SUT/CUT – System under test or Code under test
- Fixture – The environment to run the tests on the SUT/CUT
  - Pre test conditions
  - Post test cleanup
- DOC – depended-on component

Principles 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.
- Tests should be isolated and not dependent on each other

Example: E-commerce



Fixture

```
38      @Test
39      public void should_AddToCart_WhenReservedByInventory(){
40          //Arrange
41          Customer customer = new Customer();
42          InventoryService inventoryService = new InventoryRepositoryImpl();
43          Cart cart = new Cart(customer.getId(), inventoryService);
44
45          OrderItem cleaningBrushes  = new OrderItem( itemName: "Cleaning brush", quantity: 1);
46
47          //Act
48          cart.addToCart(cleaningBrushes);
49
50          //Assert
51          assertThat(cart.getCartSize(),is(equalTo( operand: 1)));
52          assertTrue(cart.fetch().contains(cleaningBrushes));
53      }
```

TECHTALKS

Code smells with Fixtures

- Duplication
- Hurts readability and understandability
- Hard to maintain

Better Fixtures

- Explicit extraction
  - Extract methods with fixture and teardown code
  - Call method in each relevant test case
- Implicit reference
  - @Before and @After annotated methods for setup/teardown

Example – Implicit Fixture

```
Ritesh Mehrotra *
@Before
public void setup() {
    customer = new Customer();
    inventoryService = mock(InventoryService.class);
}

Ritesh Mehrotra *
@After
public void teardown() {
    reset(); }

Ritesh Mehrotra *
@Test
public void should_AddToCart_WhenReservedByInventory() {
    //Arrange
    Cart cart = new Cart(customer.getId(), inventoryService);

    OrderItem cleaningBrushes = new OrderItem( itemName: "Cleaning brush",  quantity: 1);

    when(inventoryService.reserve(anyString(), anyInt())).thenReturn(ItemStatus.RESERVED);

    //Act
    cart.addToCart(cleaningBrushes);
    //Assert
    assertThat(cart.getCartSize(), is(equalTo( operand: 1)));
    assertTrue(cart.fetch().contains(cleaningBrushes));
    verify(inventoryService, times( wantedNumberOfInvocations: 1)).reserve(anyString(), anyInt());
}
```

TECHTALKS

Organizing a unit test case - AAA pattern

Arrange

- Bring SUT to a desired state before testing

Act

- Act on the SUT by executing the methods

Assert

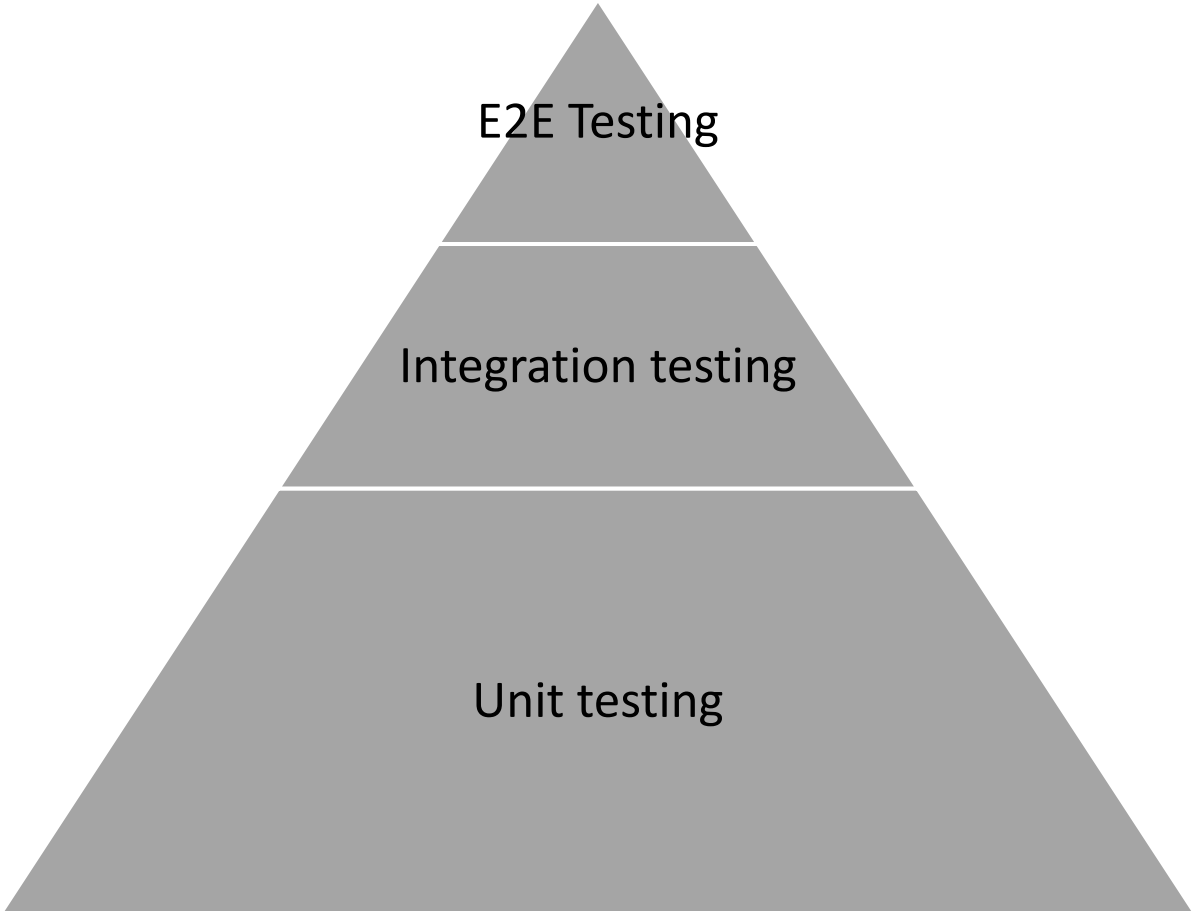
- Validate the output against expectations

```
@Test
public void shouldReturnCartTotalAsSumOfAllItemCost(){
    //Arrange
    Cart cart = new Cart();

    //Act
    cart.add(new Item( name: "iPhone 13", quantity: 1, price: 1500));
    cart.add(new Item( name: "Phone cover", quantity: 1, price: 65.50));

    //Assert
    assertThat(cart.getCartTotal(),is(equalTo( operand: 1565.50)));
}
```

TECHTALKS

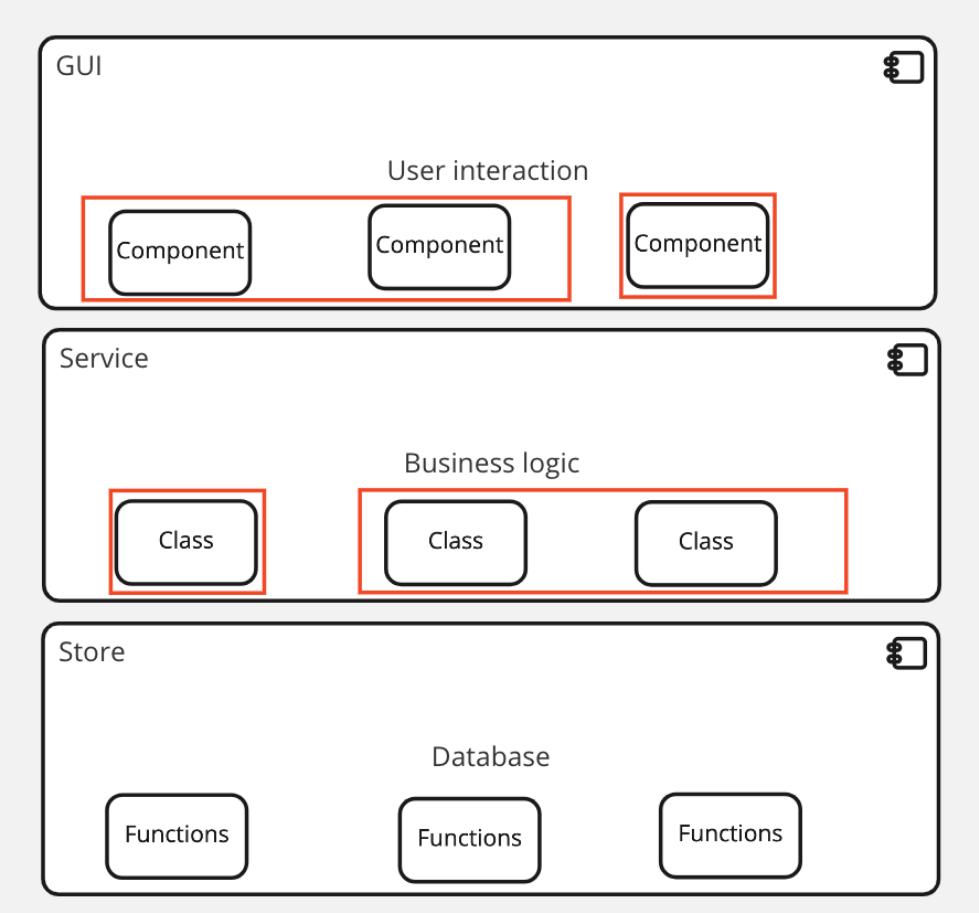
Automated Testing Strategies

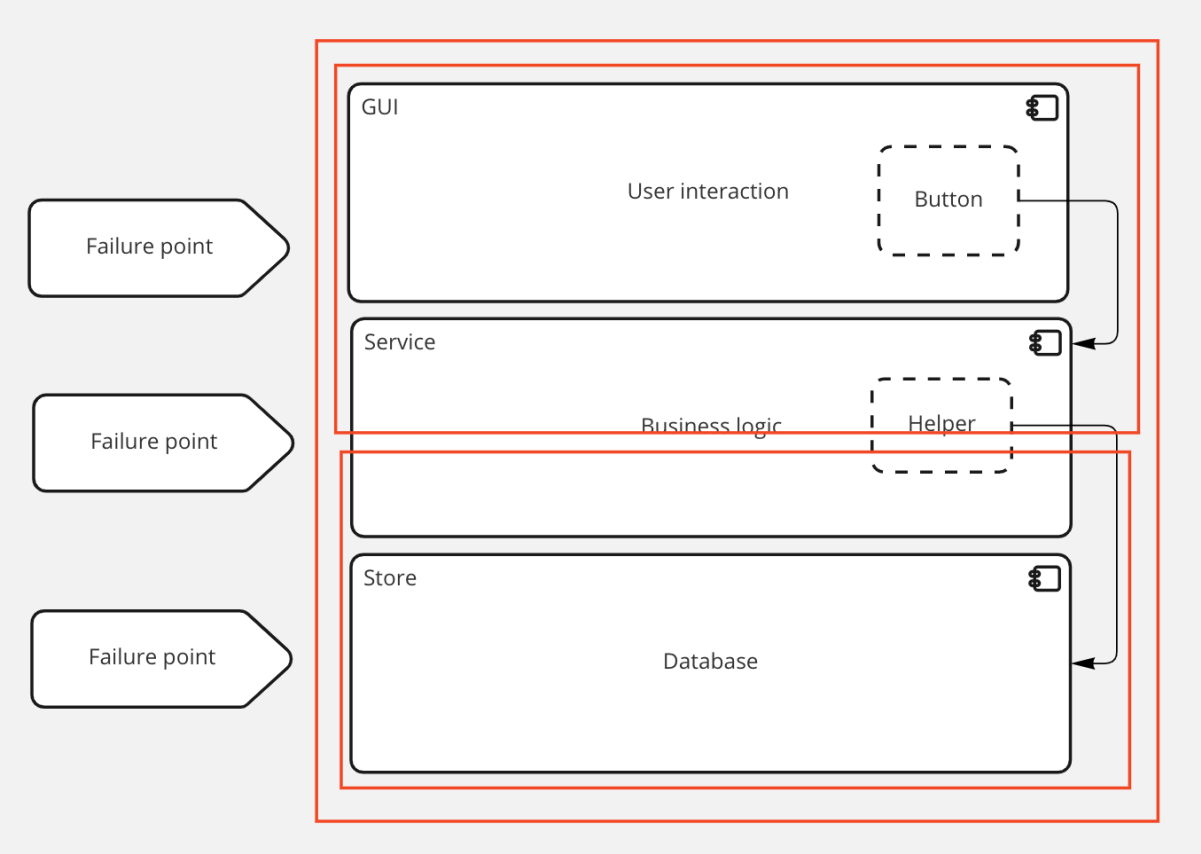
Test pyramid



- Each layer indicates a levels of granularity for testing
- The lower levels focus on isolated code behaviors
- Higher levels focus on component integration and overall system
- Unit testing is cheaper and E2E testing is costlier in terms of
  - Time to run
  - Time to change
  - Flakiness

Unit vs Integration Testing





TECHTALKS

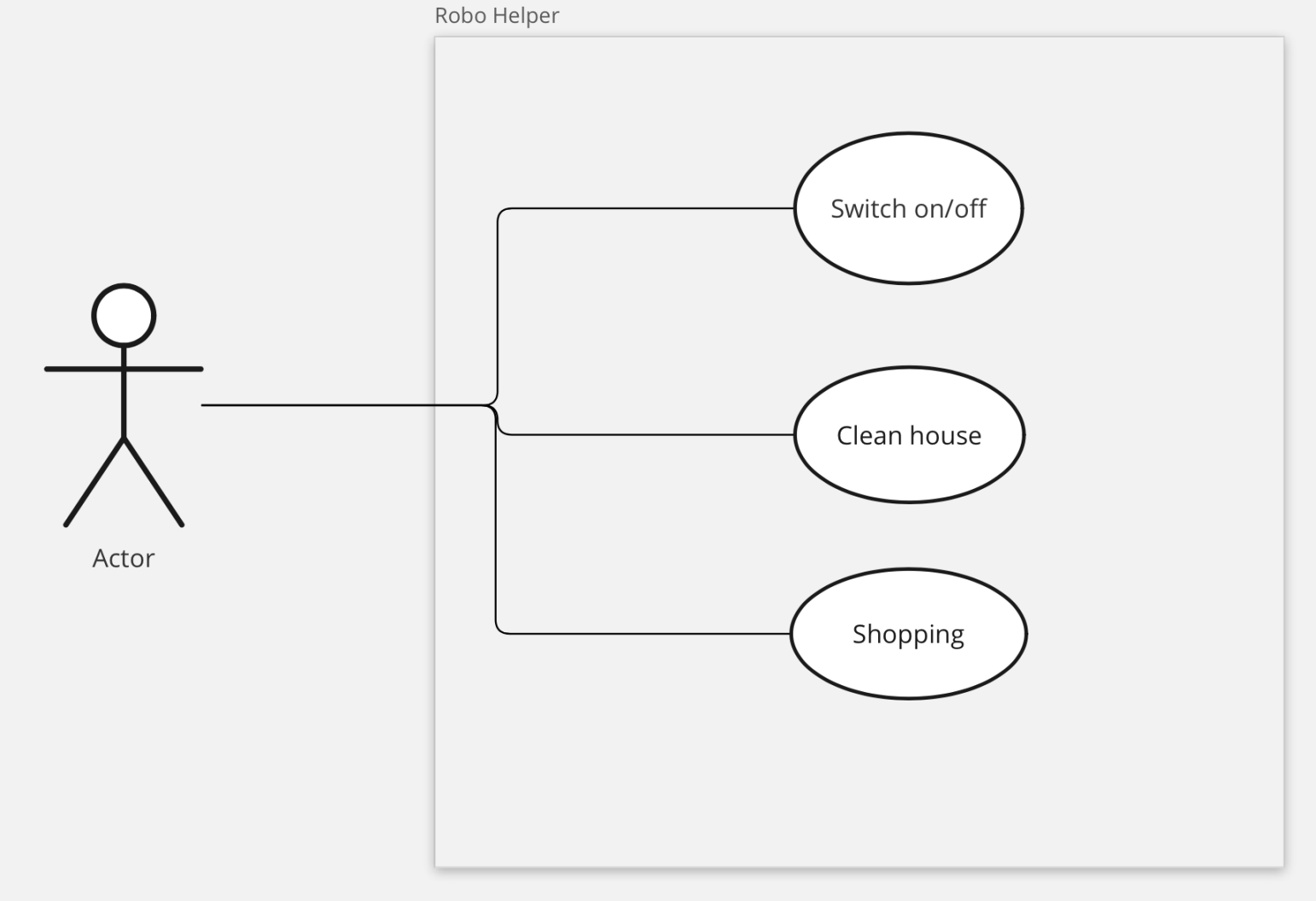
Unit Testing vs Integration Testing

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

Testability examples

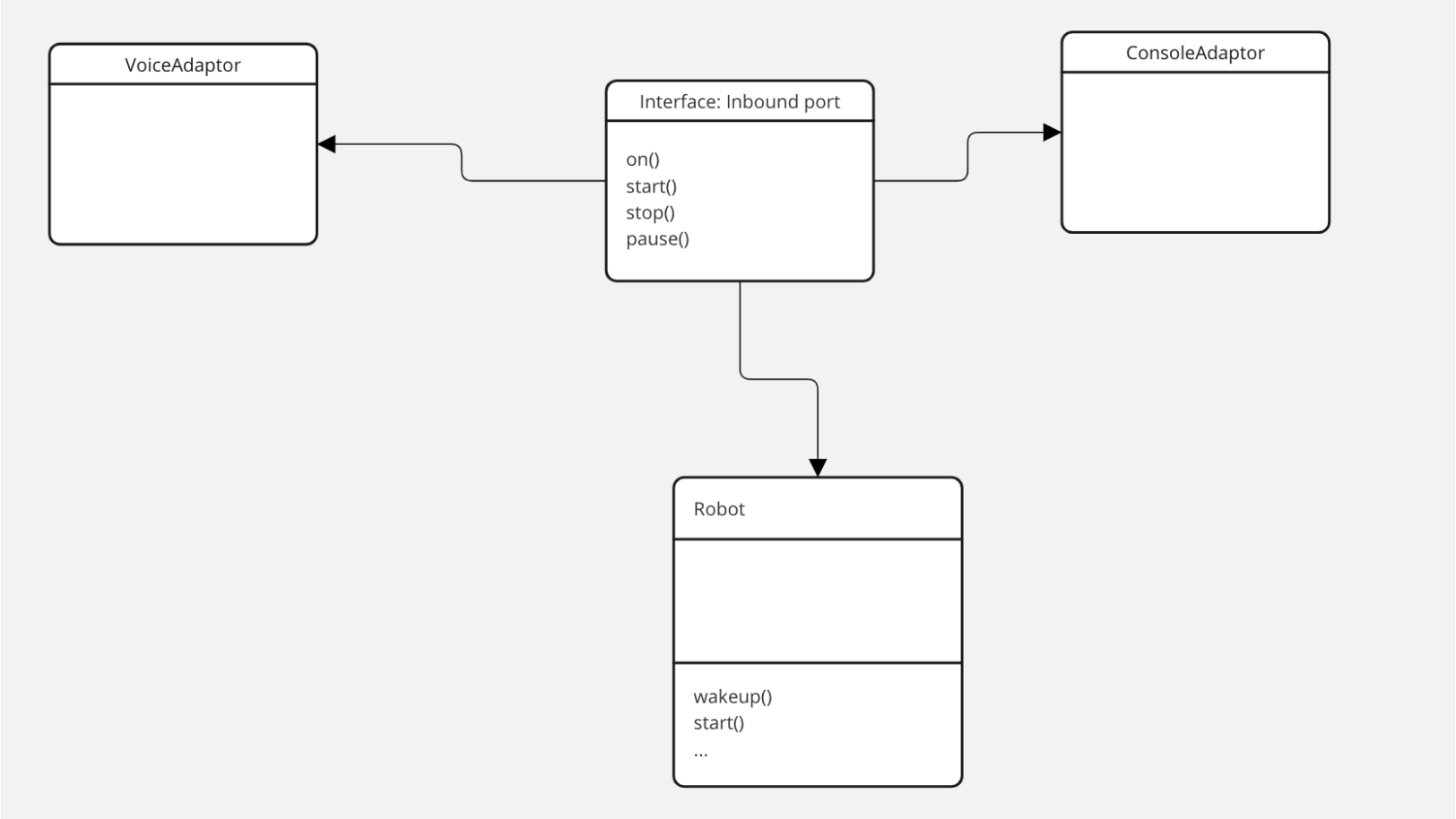
My Cleaning Robot

Use cases



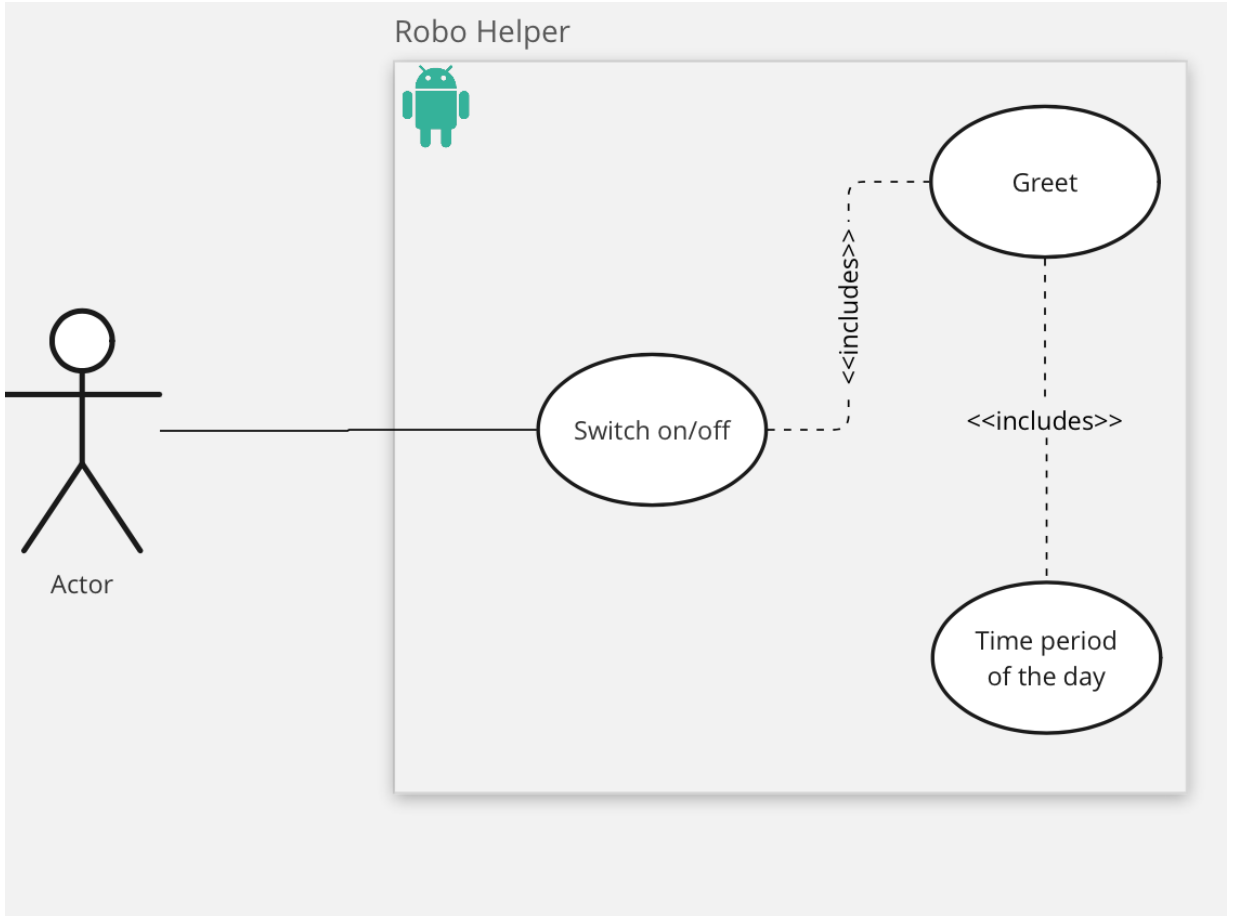
TECHTALKS

Interaction with robot



TECHTALKS

Use Case – Robo Helper Greet



TECHTALKS

Time period of the day

```
5   public class TimeOfTheDay {
6
7   @       public static String getTimePeriod() {
8               LocalDateTime dateTime = LocalDateTime.now();
9               int hour = dateTime.getHour();
10              if (hour < 12)
11                  return "Morning";
12              else if (hour > 12 && hour < 18)
13                  return "Afternoon";
14              else
15                  return "Evening";
16      }
```

- A class with static method that returns time period of the day
- Returns values “Morning”, “Afternoon” or “Evening” based on LocalDateTime

TECHTALKS

Robot class code and test

Implementation

```
16      public String start() {
17          robotStatus = RobotStatus.ON;
18          return String.format("%s %s!", "Good", TimeOfTheDay.getTimePeriod());
19      }
```

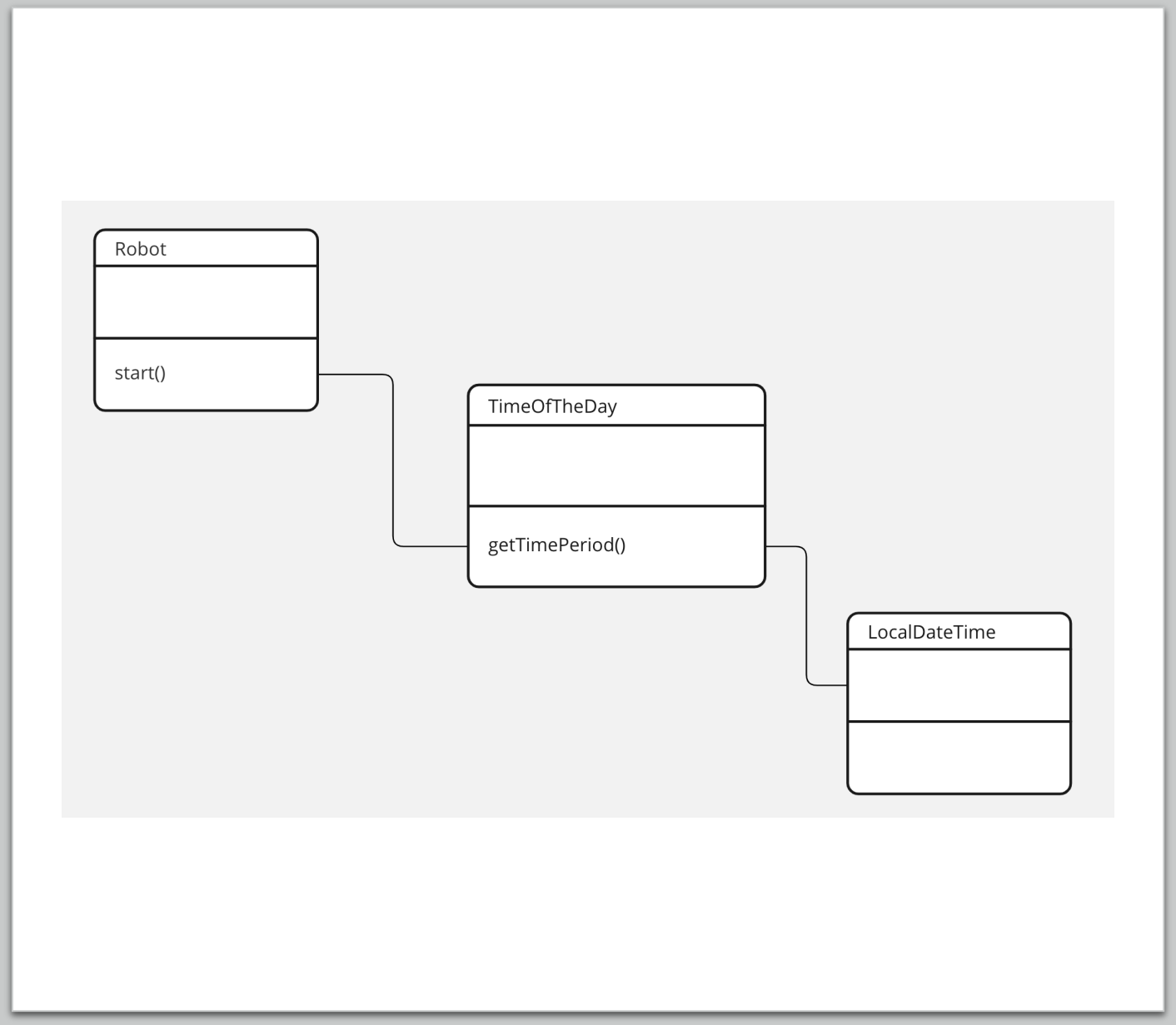
Testcase

```
32      @Test
33      public void shouldWishGoodMorningOnWakeUpCommand() {
34          Robot robot = new Robot();
35          assertEquals( expected: "Good Morning!", robot.start());
36      }
```

TECHTALKS

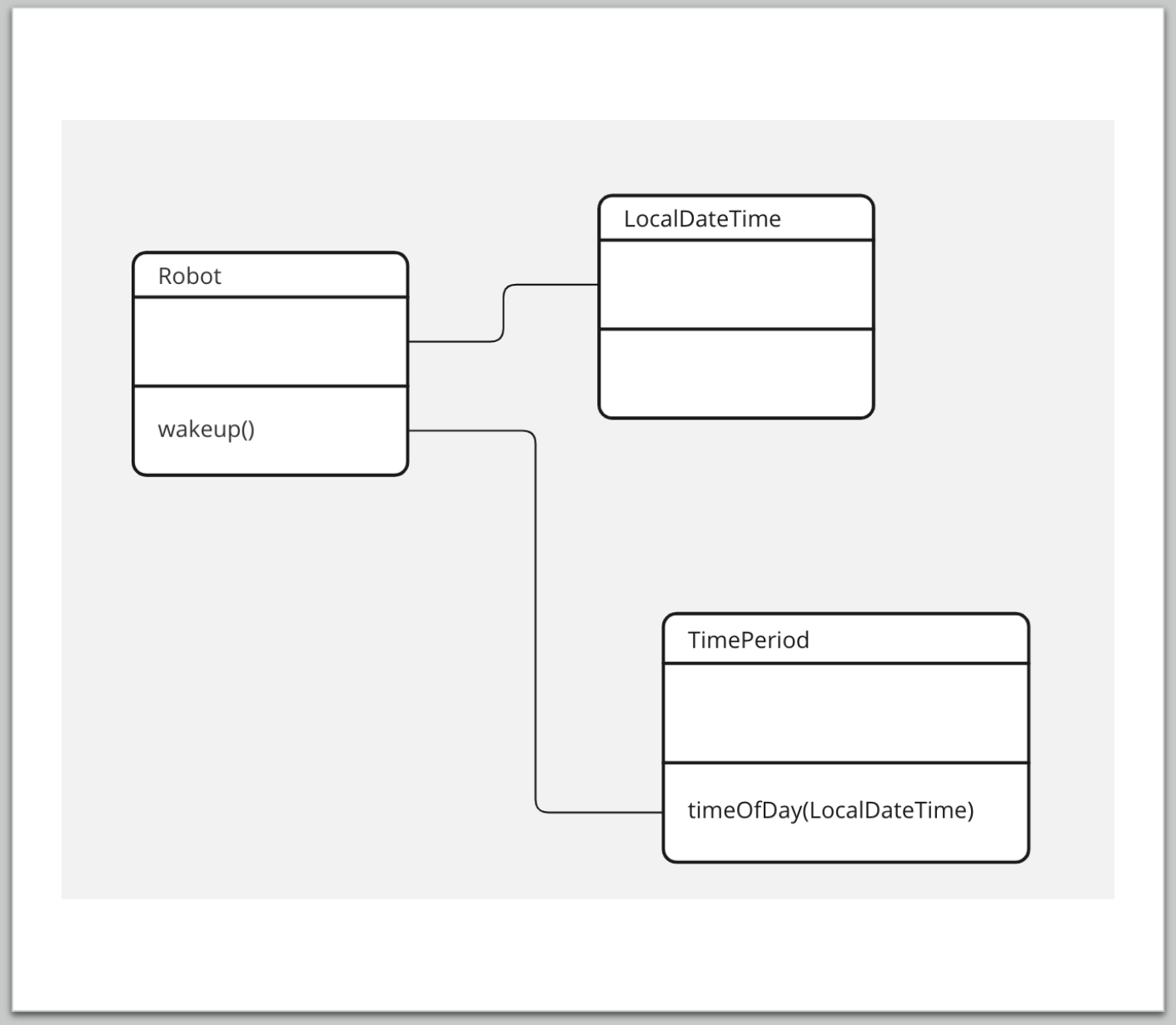
Challenges

- System time (Data source) tightly coupled within TimeOfTheDay 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



Refactored code

Implementation - Robot

```
18      public String start() {
19          robotStatus = RobotStatus.ON;
20          LocalDateTime timeNow = LocalDateTime.now();
21          return String.format("%s %s!", "Good", TimeOfTheDay.getTimePeriod(timeNow));
22      }
```

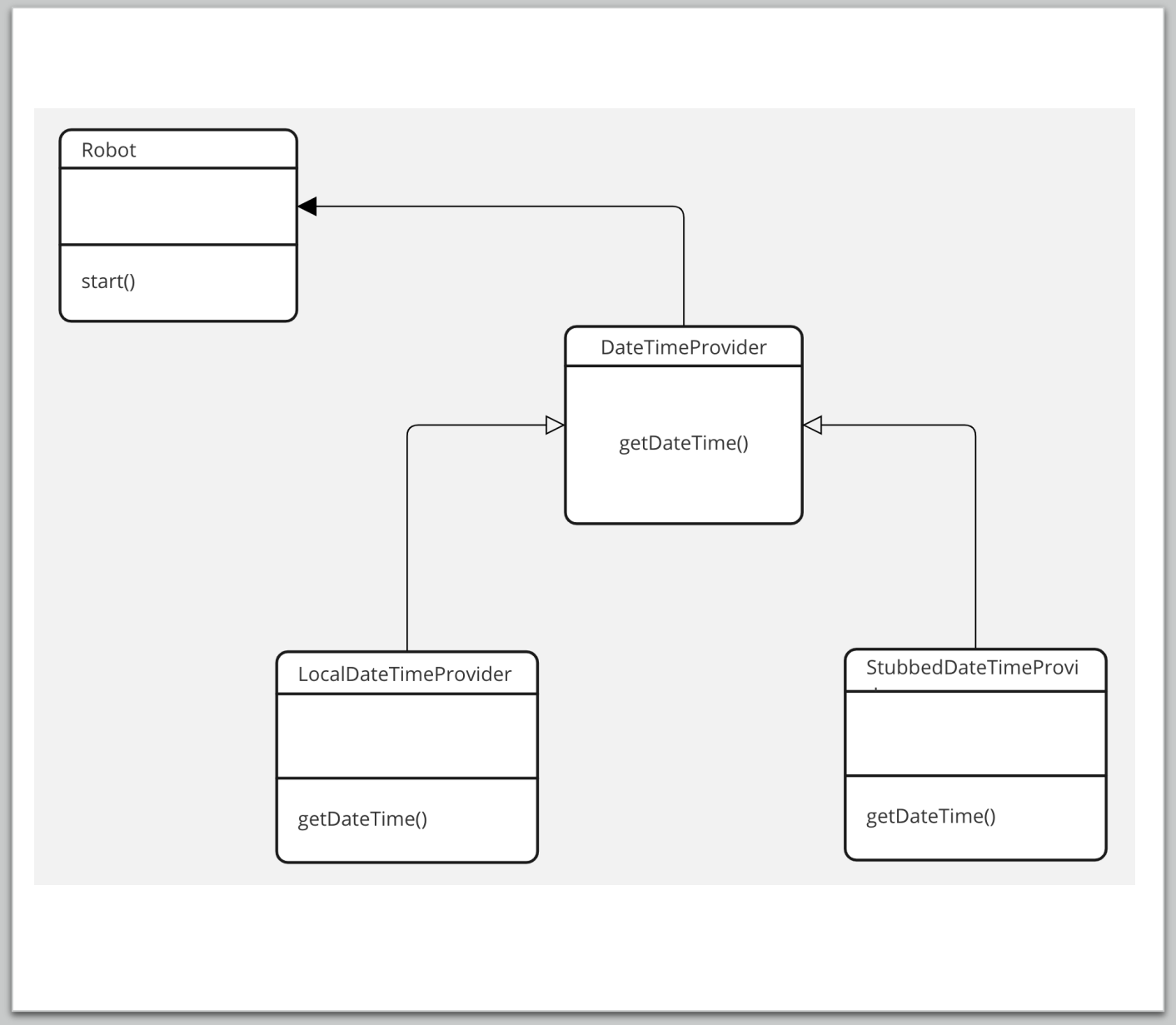
Implementation - TimeOfTheDay

```
7   @       public static String getTimePeriod(LocalDateTime dateTime) {
8               int hour = dateTime.getHour();
9               if (hour >= 9 && hour <=12)
10                  return "Morning";
11              else if (hour > 12 && hour <= 18)
12                  return "Afternoon";
13              else
14                  return "Evening";
15          }
```

TECHTALKS

Inversion of control

- Abstract concrete implementation to an interface
- Refactor to inject a stubbed implementation to Robot



Dependency injection – Robot Class

```
11      public Robot(DateTimeProvider provider) {
12          timeProvider = provider;
13          robotStatus = RobotStatus.OFF;
14      }
15
16      public String start() {
17          robotStatus = RobotStatus.ON;
18          LocalDateTime timeNow = timeProvider.getDateTime();
19          return String.format("%s %s!", "Good", TimeOfTheDay.getTimePeriod(timeNow));
20      }
```

TECHTALKS

DateTimeProvider implementations

```
5  public interface DateTimeProvider {
       3 implementations
6      LocalDateTime getDateTime();
7  }
```

```
5  public class LocalDateTimeProvider implements DateTimeProvider {
6      @Override
7      public LocalDateTime getDateTime() {
8          return LocalDateTime.now();
9      }
```

```
5  public class StubbedDateTimeProvider implements DateTimeProvider {
6      @Override
7      public LocalDateTime getDateTime() {
8          return LocalDateTime.of( year: 2022, month: 8, dayOfMonth: 2, hour: 10, minute: 0);
9      }
10 }
```

TECHTALKS

Testing Robot Greeting

Real implementation

```
9   public class RobotInitializer {
10
11      public String activateRobot() {
12          DateTimeProvider provider = new LocalDateTimeProvider();
13          Robot robot = new Robot(provider);
14          return robot.start();
15      }
16
17  }
```

Test implementation

```
24      @Test
25      public void shouldWishGoodMorningOnWakeUpCommand() {
26          DateTimeProvider timeProvider = new StubbedDateTimeProvider();
27          Robot robot = new Robot(timeProvider);
28          assertEquals( expected: "Good Morning!", robot.start());
29      }
```

Test doubles

Types of test doubles

Stubs

Dummy

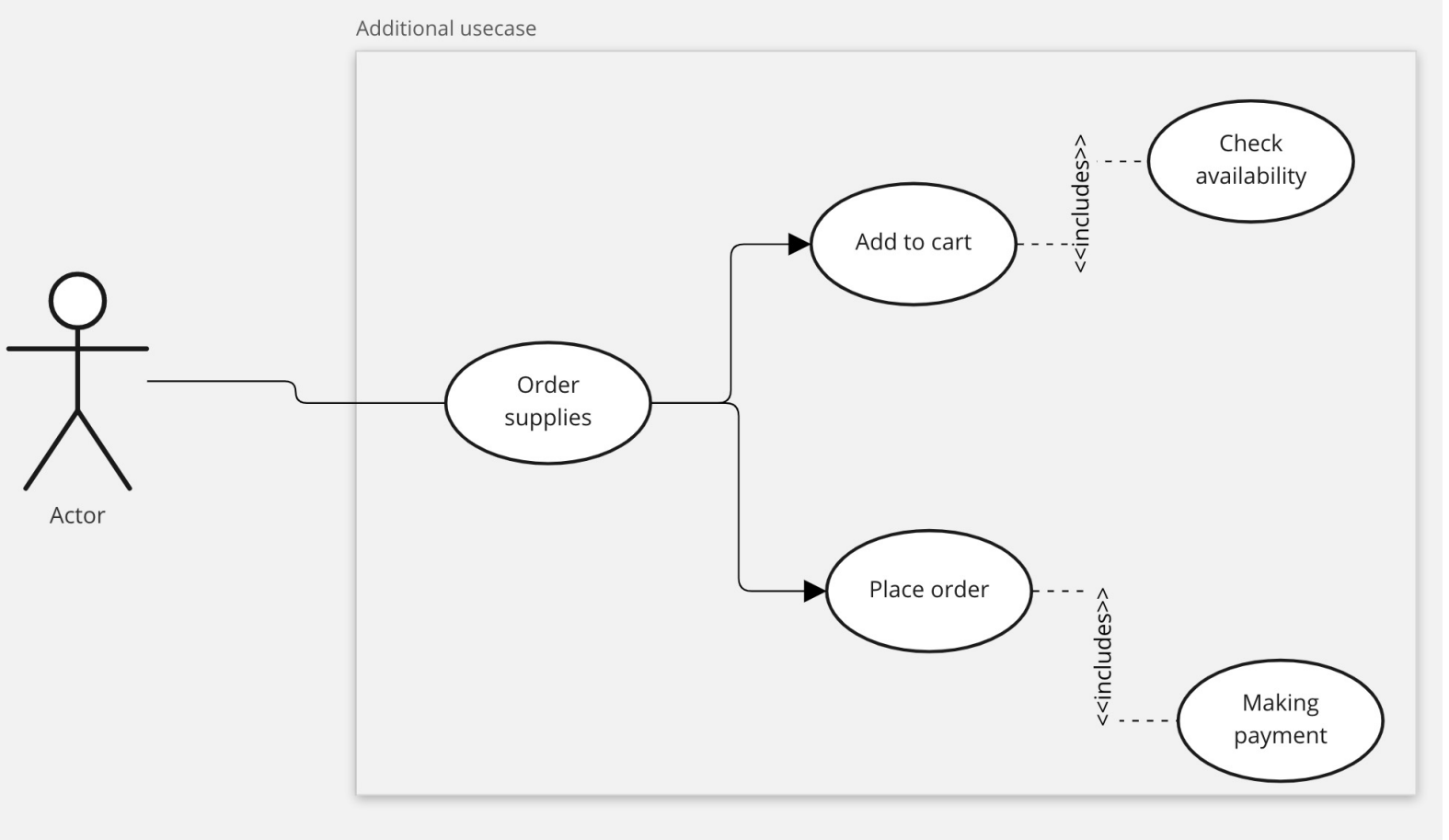
Fakes

Mock

Spies

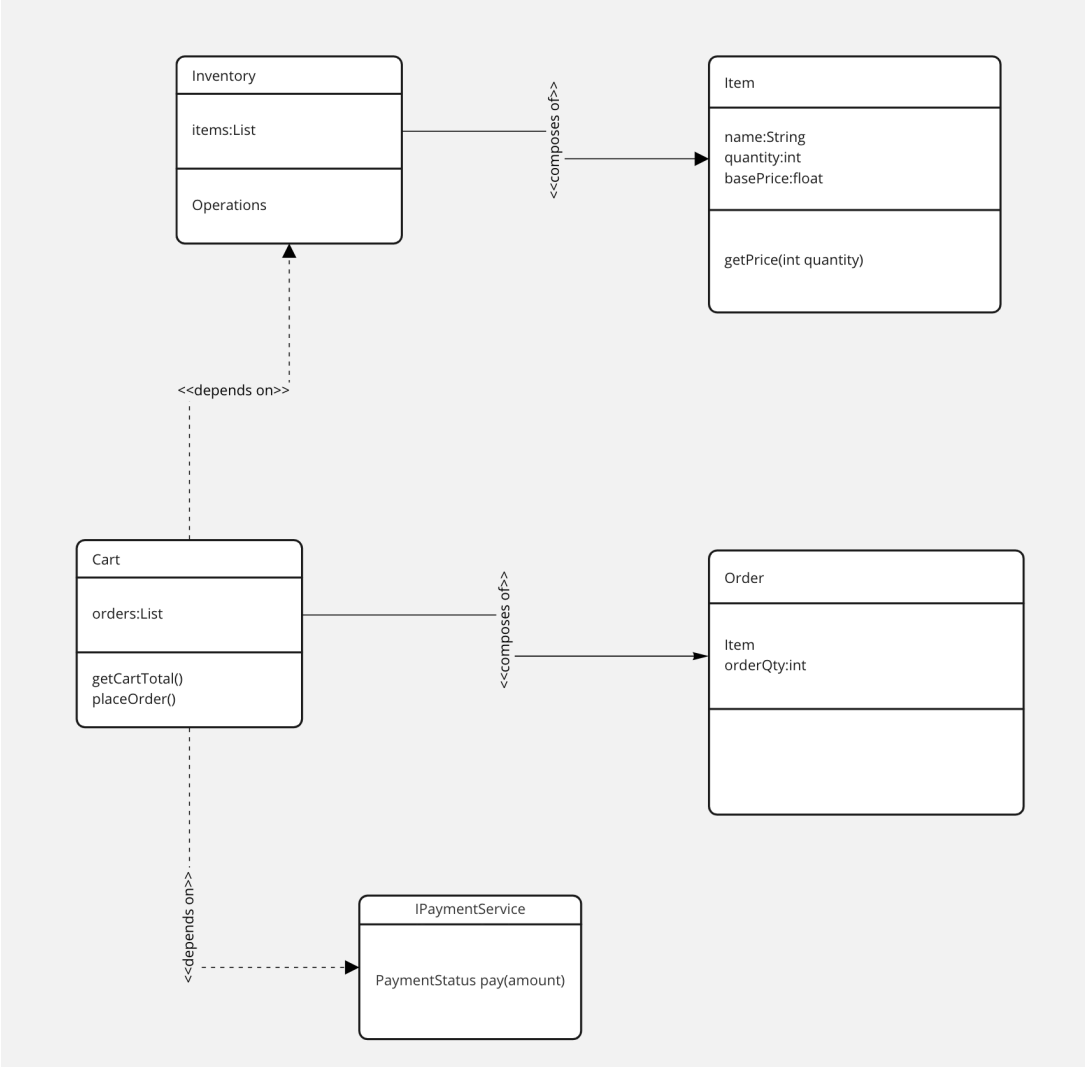
TECHTALKS

Use case – Order robot supplies



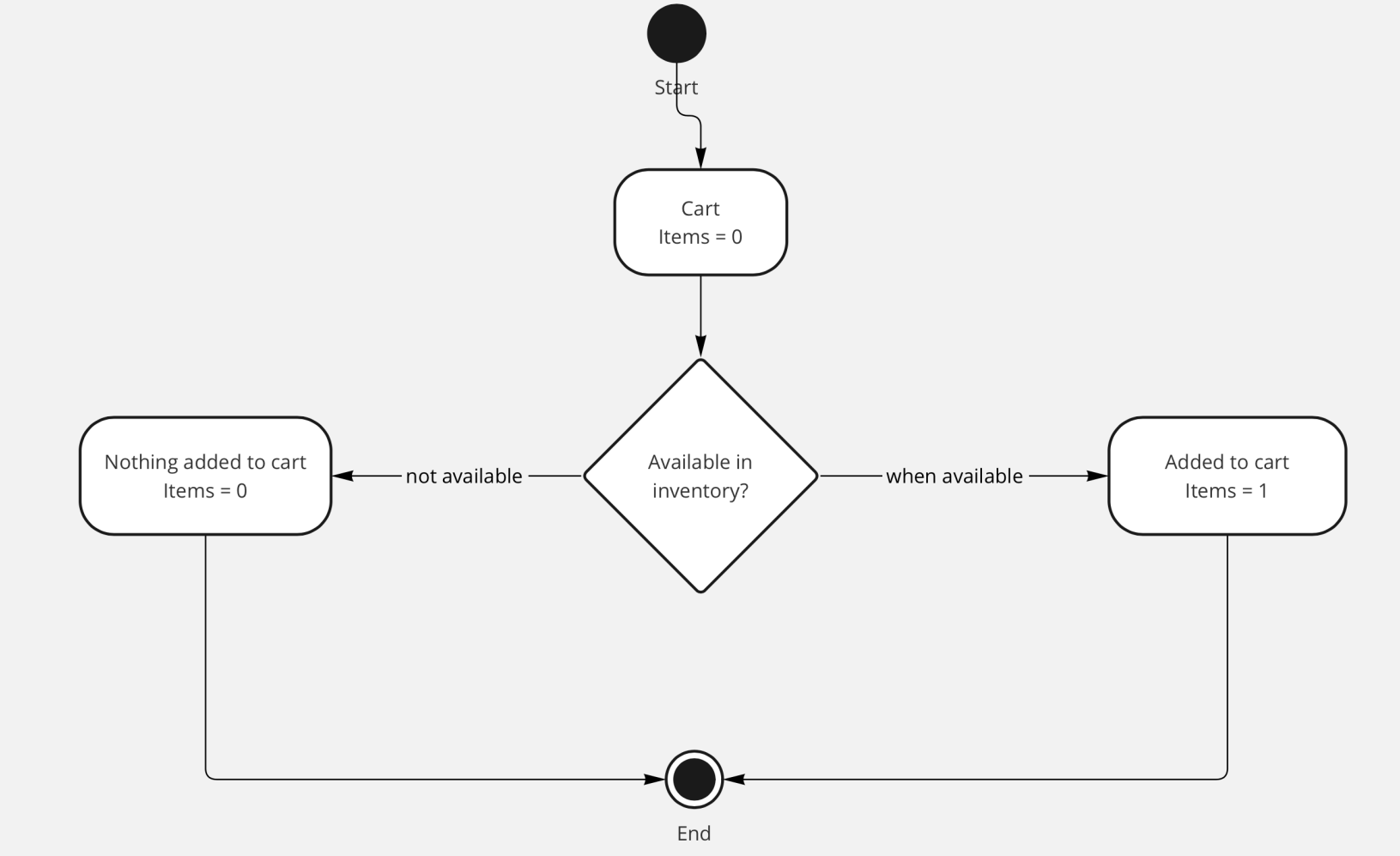
TECHTALKS

Code Structure



TECHTALKS

Scenarios – Cart behavior



TECHTALKS

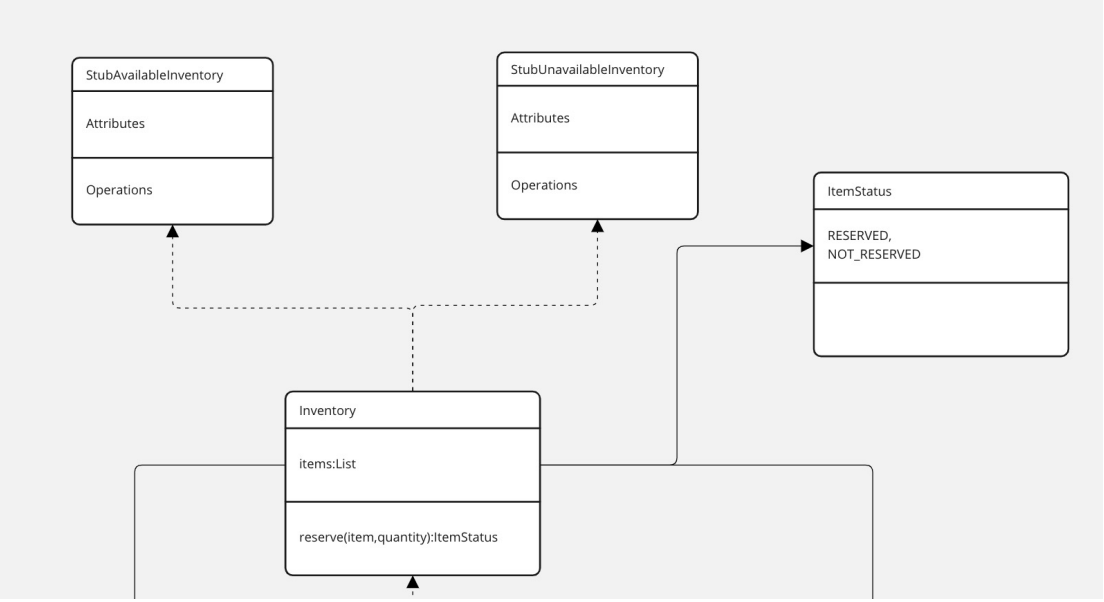
Cart class

```
6   public class Cart {
7       List<OrderItem> orders = new ArrayList<>();
8       InventoryService inventoryService;
9       PaymentService paymentService;
10
11      public Cart(InventoryService inventoryService, PaymentService paymentService){
12          this.inventoryService=inventoryService;
13          this.paymentService=paymentService;
14      }
15
16  @   public void addToCart(OrderItem orderItem){
17          ItemStatus inventoryResponse=inventoryService.reserve(orderItem.getItemName(),orderItem.getQuantity());
18          if(inventoryResponse.equals(ItemStatus.RESERVED)){
19              orders.add(orderItem);
20          }
21      }
22
23      public int getCartSize(){
24          return orders.size();
25      }
26
27
28      public List<OrderItem> fetch(){
29          return orders;
30      }
31  }
```

TECHTALKS

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




```java
public class StubAvailableInventoryService implements InventoryService {

    @Override
    public ItemStatus reserve(String itemName, int quantity) {
        return ItemStatus.RESERVED;
    }
}
```

```java
public class StubUnavailableInventoryService implements InventoryService {

    @Override
    public ItemStatus reserve(String itemName, int quantity) {
        return ItemStatus.NOT_RESERVED;
    }
}
```

Cart test with stubs

With available stock

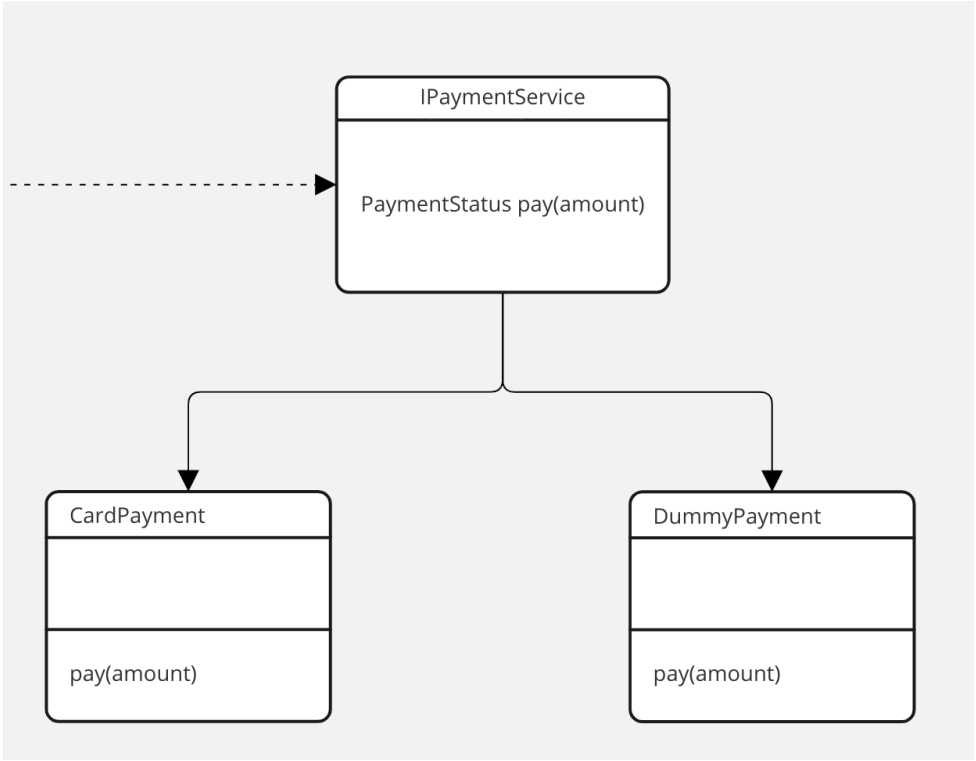
```
12      @Test
13      public void should_AddToCart_WhenQuantityAvailableInInventory(){
14          //Arrange
15          InventoryService inventoryService = new StubAvailableInventoryService();
16          PaymentService paymentService = new DummyPaymentService();
17          Cart cart = new Cart(inventoryService,paymentService);
18          OrderItem orderItem  = new OrderItem( itemName: "Cleaning brush", quantity: 1);
19
20          //Act
21          cart.addToCart(orderItem);
22
23          //Assert
24          assertThat(cart.getCartSize(),is(equalTo( operand: 1)));
25      }
```

Without stock

```
28      @Test
29      public void shouldNot_AddToCart_WhenQuantityUnavailableInInventory(){
30          //Arrange
31          InventoryService inventoryService = new StubUnavailableInventoryService();
32          PaymentService paymentService = new DummyPaymentService();
33          Cart cart = new Cart(inventoryService,paymentService);
34          OrderItem orderItem  = new OrderItem( itemName: "Cleaning brush", quantity: 1);
35
36          //Act
37          cart.addToCart(orderItem);
38
39          //Assert
40          assertThat(cart.getCartSize(),is(equalTo( operand: 0)));
41      }
```

Dummy

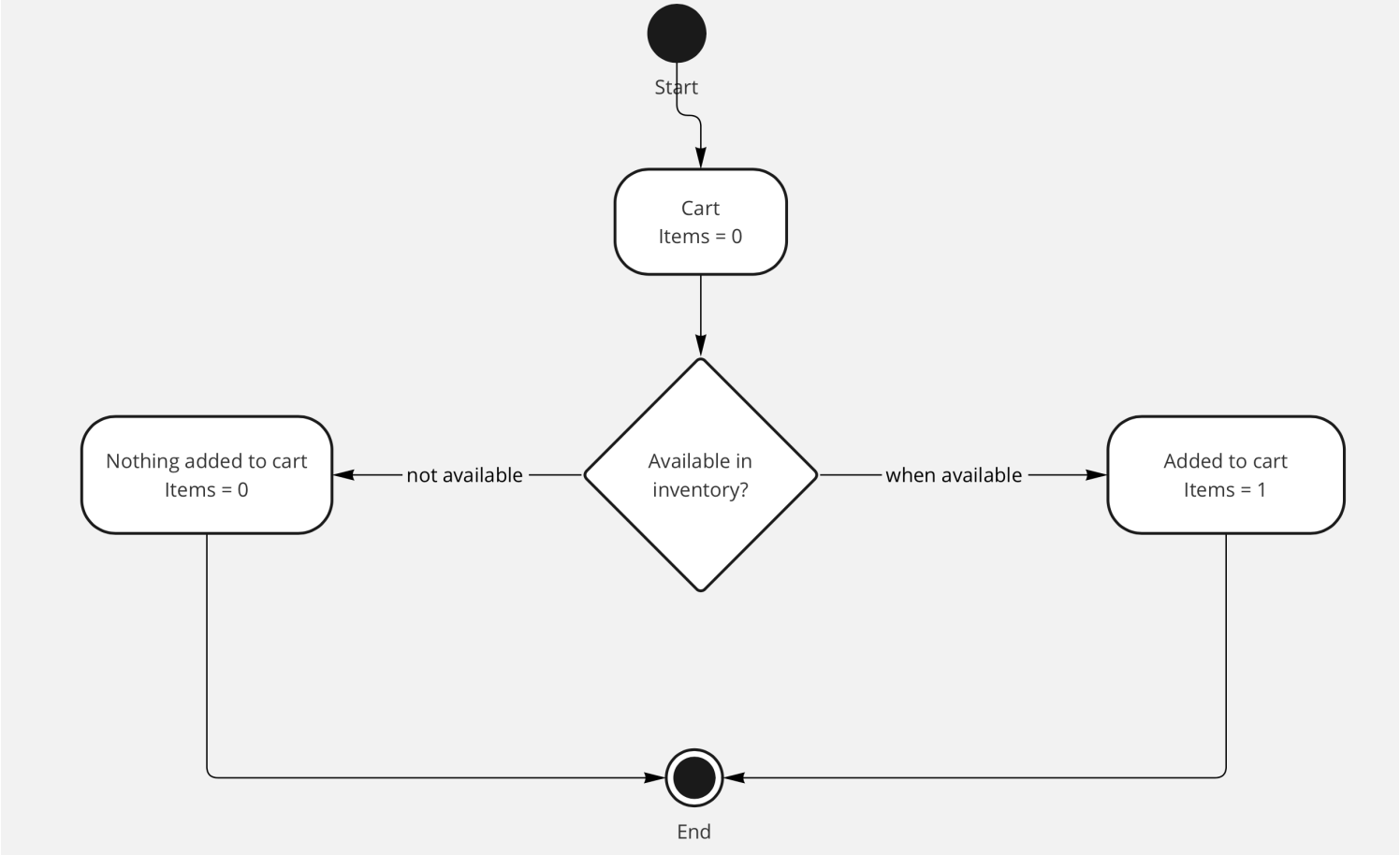
A filler object to satisfy structural needs of code under test. Used as a test helper and doesn't impact the behavior of code under test.




```java
public class DummyPaymentService implements PaymentService {
    @Override
    public PaymentStatus pay(float amount) {
        return null;
    }
}
```

```java
    @Test
    public void should_AddToCart_WhenQuantityAvailableInInventory(){
        //Arrange
        InventoryService inventoryService = new StubAvailableInventoryService();
        PaymentService paymentService = new DummyPaymentService();
        Cart cart = new Cart(inventoryService,paymentService);
        OrderItem orderItem  = new OrderItem( itemName: "Cleaning brush", quantity: 1);

        //Act
        cart.addToCart(orderItem);

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

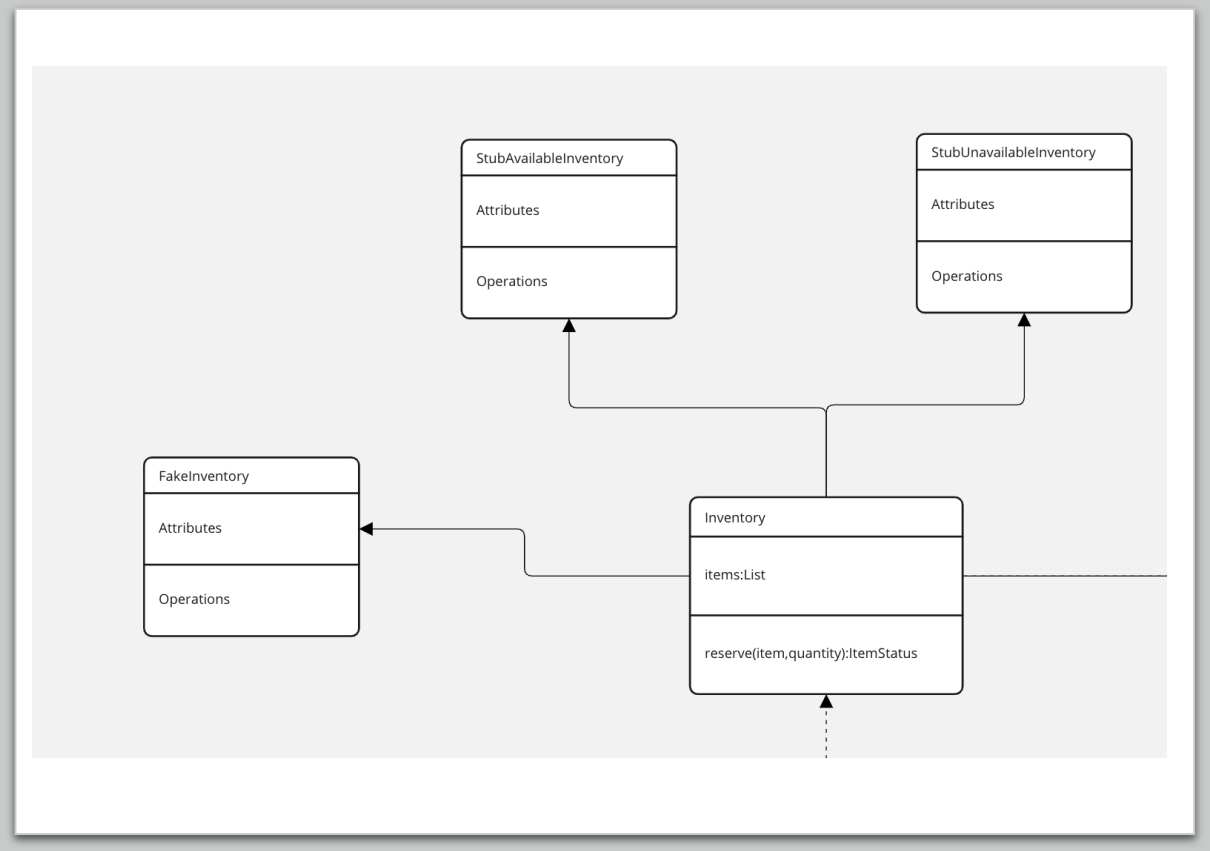
Scenarios – Cart behavior with multiple items



TECHTALKS

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




```java
public class FakeInventoryService implements InventoryService {

    List<Item> items;
    public FakeInventoryService() {
        items = new ArrayList<>();
        this.initialize();
    }

    private void initialize() {
        Item cleaningBrushes = new Item( name: "Cleaning brush", basePrice: 5, totalQuantity: 100);
        Item mops = new Item( name: "Mop", basePrice: 25, totalQuantity: 100);
        Item filters = new Item( name: "filter", basePrice: 15, totalQuantity: 100);

        items.add(cleaningBrushes);
        items.add(mops);
        items.add(filters);
    }


    @Override
    public ItemStatus reserve(String itemName, int quantity) {
        Optional<Item> filteredItem =
                items.stream().filter(item -> item.getName().equals(itemName) && item.getTotalQuantity() >= quantity).findFirst();

        //Fake implementation: No logic needed to deduct quantity from inventory
        //doNothing();

        return (filteredItem.isPresent()) ? ItemStatus.RESERVED : ItemStatus.NOT_RESERVED;
    }
}
```

Cart test with Fake inventory

```
48      @Test
49      public void should_AddToCart_AllItemsAvailableInInventory(){
50          InventoryService inventoryService = new FakeInventoryService();
51          PaymentService paymentService = new DummyPaymentService();
52          Cart cart = new Cart(inventoryService,paymentService);
53
54          OrderItem cleaningBrushes = new OrderItem( itemName: "Cleaning brush", quantity: 2);
55          OrderItem charger  = new OrderItem( itemName: "Charger", quantity: 1);
56
57          //Act
58          cart.addToCart(cleaningBrushes);
59          cart.addToCart(charger);
60
61          //Assert
62          assertThat(cart.getCartSize(),is(equalTo( operand: 1)));
63          assertTrue(cart.fetch().contains(cleaningBrushes));
64      }
```

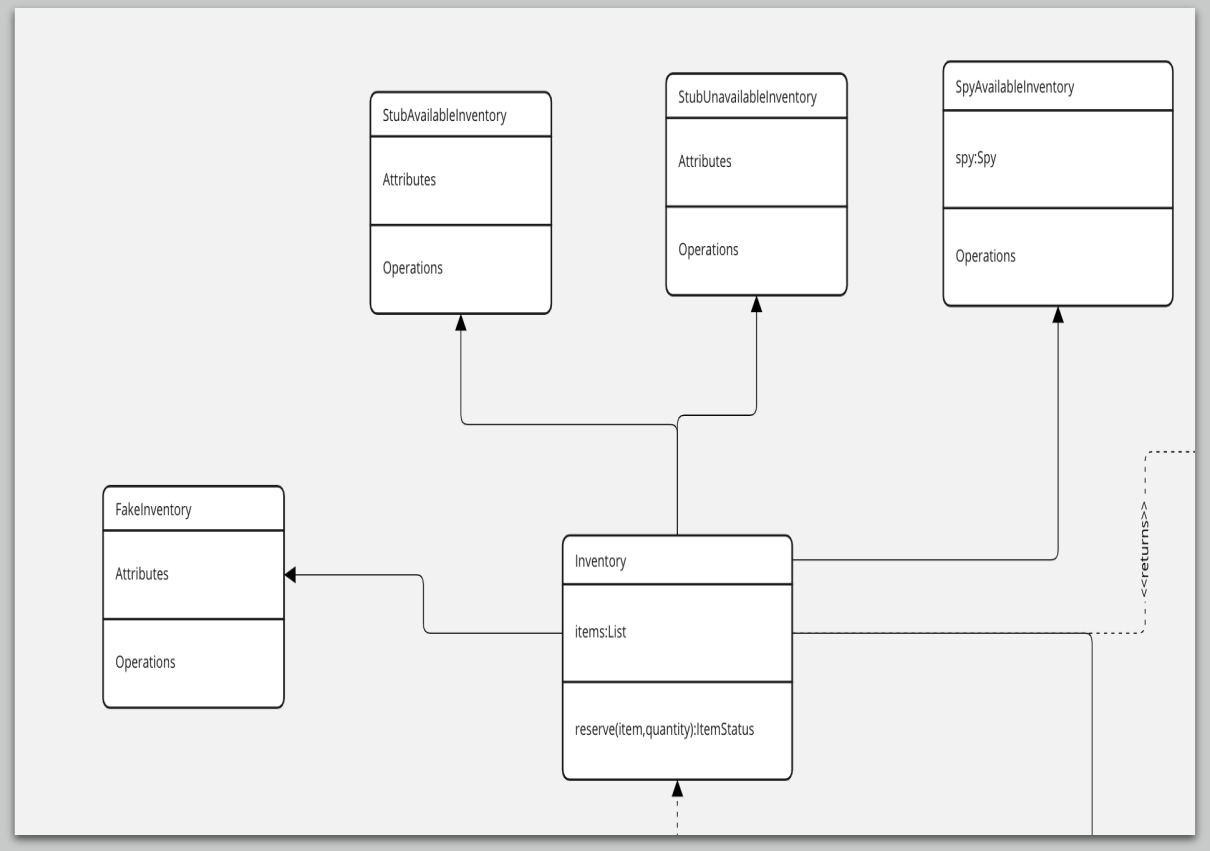
TECHTALKS

Scenario – Invalid cart order

- Adding order to cart
  - When item name is empty or when quantity is not greater than ZERO
    - Don’t call inventory service
    - Don’t add to cart

Spy

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




```java
public class SpyAvailableInventoryService implements InventoryService {
    Spy spy;
    public SpyAvailableInventoryService(Spy spy){
        this.spy = spy;
    }

    @Override
    public ItemStatus reserve(String itemName, int quantity) {
        spy.incrementTimes();
        return ItemStatus.RESERVED;
    }

}
```

```java
@Data
public class Spy {
    int timesCalled;

    public void incrementTimes() {
        timesCalled++;
    }
}
```

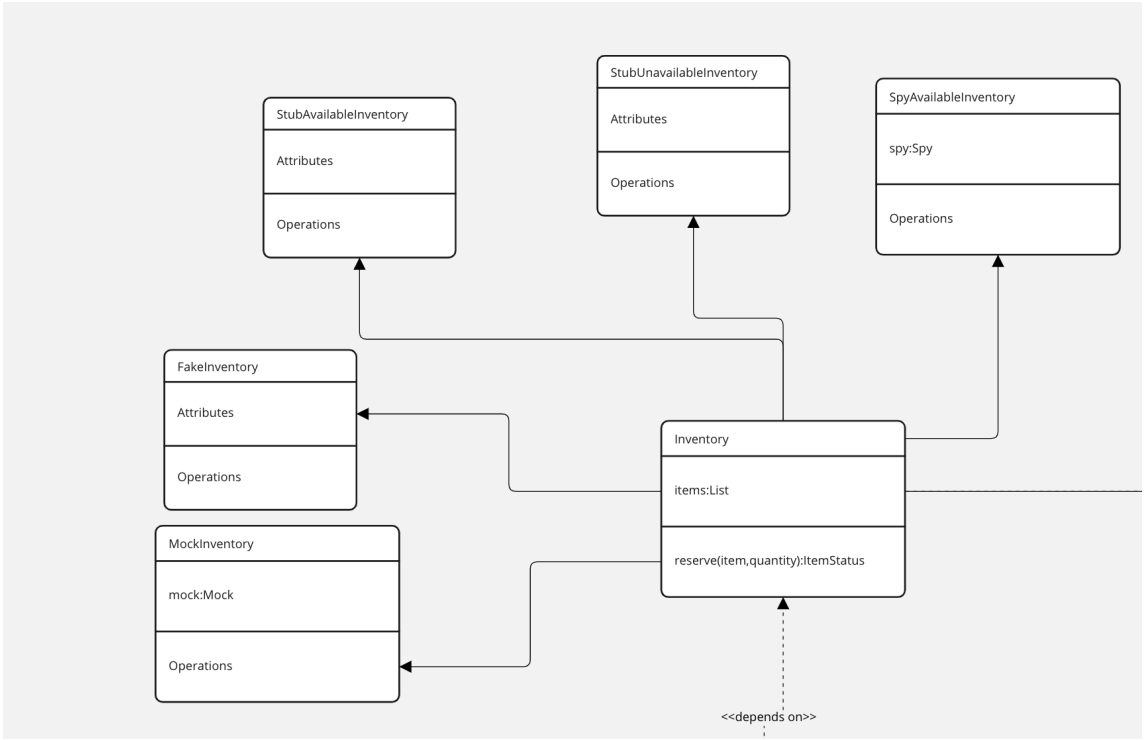
Cart test with Spy inventory

```
67      @Test
68      public void should_AddToCart_AllItemsAvailableInInventoryWithSpy(){
69          Spy spy = new Spy();
70          InventoryService inventoryService = new SpyAvailableInventoryService(spy);
71          PaymentService paymentService = new DummyPaymentService();
72          Cart cart = new Cart(inventoryService,paymentService);
73
74          OrderItem cleaningBrushes = new OrderItem( itemName: "Cleaning brush", quantity: 2);
75          OrderItem filters = new OrderItem( itemName: "Filter", quantity: 3);
76          OrderItem charger  = new OrderItem( itemName: "Charger", quantity: 0);
77          OrderItem unknownItem  = new OrderItem( itemName: null, quantity: 100);
78
79          //Act
80          cart.addToCart(cleaningBrushes);
81          cart.addToCart(filters);
82          cart.addToCart(charger);
83          cart.addToCart(unknownItem);
84
85          //Assert
86          assertThat(cart.getCartSize(),is(equalTo( operand: 2)));
87          assertTrue(cart.fetch().contains(cleaningBrushes));
88          assertTrue(cart.fetch().contains(filters));
89          assertThat(spy.getTimesCalled(),is(equalTo( operand: 2)));
90      }
```

TECHTALKS

Mock

Mock is a spy that verifies the interactions with caller object and fails if it doesn't meet expectations




```java
public class MockInventoryService implements InventoryService {
    Mock mock;
    public MockInventoryService(Mock mock) { this.mock = mock; }

    @Override
    public ItemStatus reserve(String itemName, int quantity) {
        mock.incrementTimes();
        mock.setMethodNameActual("reserve");
        return ItemStatus.RESERVED;
    }
```

Mock class

```java
@Data
public class Mock {
    int times;
    private int timesExpected;
    String methodNameActual;
    private String methodNameExpected;

    public Mock calledTimes(int timesExpectation) {
        this.timesExpected=timesExpectation;
        return this;
    }

    public Mock expectMethod(String methodNameExpected) {
        this.methodNameExpected = methodNameExpected;
        return this;
    }

    public void incrementTimes() {
        times++;
    }

    public void verify() {
        if (!methodNameActual.equals(methodNameExpected))
            throw new IllegalArgumentException(String.format("Expected %s method to be called, but it wasn't", methodNameExpected));

        if (times != timesExpected)
            throw new IllegalArgumentException(String.format("Expected to be called %d times, but was called %d times", timesExpected, times));

    }
}
```

TECHTALKS

Cart test with Mock inventory

```
93      @Test
94      public void should_AddToCart_AllItemsAvailableInInventoryWithMock(){
95          //Arrange
96          Mock mock = new Mock();
97          InventoryService inventoryService = new MockInventoryService(mock);
98          PaymentService paymentService = new DummyPaymentService();
99          Cart cart = new Cart(inventoryService,paymentService);
100
101         mock.expectMethod( methodNameExpected: "reserve").calledTimes( timesExpectation: 2);
102
103         OrderItem cleaningBrushes = new OrderItem( itemName: "Cleaning brush", quantity: 2);
104         OrderItem filters = new OrderItem( itemName: "Filter", quantity: 3);
105         OrderItem charger  = new OrderItem( itemName: "Charger", quantity: 0);
106
107         //Act
108         cart.addToCart(cleaningBrushes);
109         cart.addToCart(filters);
110         cart.addToCart(charger);
111
112         //Assert
113         mock.verify();
114     }
```

TECHTALKS

Some Test frameworks

Java – Mockito, EasyMock

React - Jest

Javascript – Jasmine, Sinon

Python – unittest.mock

C#- Moq

Examples

Mockito mock

```
31      @Test
32      public void should_AddToCart_WhenQuantityAvailableInInventory_WithMockito(){
33          //Arrange
34          InventoryService inventoryService = mock(InventoryService.class);
35          PaymentService paymentService = mock(PaymentService.class);
36          Cart cart = new Cart(inventoryService,paymentService);
37          OrderItem cleaningBrushes  = new OrderItem( itemName: "Cleaning brush", quantity: 1);
38
39          when(inventoryService.reserve(cleaningBrushes.getItemName(),cleaningBrushes.getQuantity())).thenReturn(ItemStatus.RESERVED);
40          //Act
41          cart.addToCart(cleaningBrushes);
42
43          //Assert
44          assertThat(cart.getCartSize(),is(equalTo( operand: 1)));
45          assertTrue(cart.fetch().contains(cleaningBrushes));
46          verify(inventoryService,times( wantedNumberOfInvocations: 1));
47      }
```

Alternative for expectation

when(inventoryService.reserve(anyString(),anyInt())).thenReturn(ItemStatus.RESERVED);

Alternative for verification

verify(inventoryService).reserve(cleaningBrushes.getItemName(),cleaningBrushes.getQuantity());

TECHTALKS

Mockito mock facts

- It’s a test double, unless you want to call real method
- Stubbing is integrated with then methods

```
50      @Test
51      public void should_AddToCart_WhenQuantityAvailableInInventory_WithMockitoAndReal(){
52          //Arrange
53          InventoryService inventoryService = mock(StubAvailableInventoryService.class);
54          PaymentService paymentService = mock(PaymentService.class);
55          Cart cart = new Cart(inventoryService,paymentService);
56          OrderItem cleaningBrushes  = new OrderItem( itemName: "Cleaning brush", quantity: 1);
57
58          when(inventoryService.reserve(anyString(),anyInt())).thenCallRealMethod();
59          //Act
60          cart.addToCart(cleaningBrushes);
61
62          //Assert
63          assertThat(cart.getCartSize(),is(equalTo( operand: 1)));
64          assertTrue(cart.fetch().contains(cleaningBrushes));
65          verify(inventoryService,times( wantedNumberOfInvocations: 1));
66      }
```

TECHTALKS

Mockito Spy

- Represents the concept of partial mock
- It calls real method, unless you want to mock

```
@Test
public void should_AddToCart_WhenQuantityAvailableInInventory_WithMockitoSpy(){
    //Arrange
    InventoryService inventoryService = spy(StubAvailableInventoryService.class);
    PaymentService paymentService = mock(PaymentService.class);
    Cart cart = new Cart(inventoryService,paymentService);
    OrderItem cleaningBrushes  = new OrderItem( itemName: "Cleaning brush", quantity: 1);

    //Act
    cart.addToCart(cleaningBrushes);

    //Assert
    assertThat(cart.getCartSize(),is(equalTo( operand: 1)));
    assertTrue(cart.fetch().contains(cleaningBrushes));
    verify(inventoryService,times( wantedNumberOfInvocations: 1));
}
```

TECHTALKS

Mockito Spy with Stubbed return value

```java
@Test
public void shouldNot_AddToCart_WithMockitoSpy(){
    //Arrange
    InventoryService inventoryService = spy(InventoryService.class);
    PaymentService paymentService = mock(PaymentService.class);
    Cart cart = new Cart(inventoryService,paymentService);
    OrderItem cleaningBrushes  = new OrderItem( itemName: "Cleaning brush", quantity: 1);

    doReturn(ItemStatus.NOT_RESERVED).when(inventoryService).reserve(anyString(),anyInt());
    //Act
    cart.addToCart(cleaningBrushes);

    //Assert
    assertThat(cart.getCartSize(),is(equalTo( operand: 0)));
    verify(inventoryService,times( wantedNumberOfInvocations: 0));
}
```

TECHTALKS

Conclusion

- Unit testing helps to keep code maintainable and understandable
- Helps to quickly validate the code expectations and hence ensuring quality
- Test double strategies help to replace the behaviors of DOC and run tests in isolation

Thank you

Connect With Me

TECHTALKS