

Patterns in Software Engineering

09 Testing Patterns II

Stephan Krusche

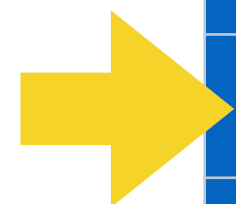


16 January 2023
Technical University of Munich



Course schedule

#	Date	Subject
	17.10.22	No lecture, repetition week (self-study)
1	24.10.22	Introduction
	31.10.22	No lecture, repetition week (self-study)
2	07.11.22	Design Patterns I
3	14.11.22	Design Patterns II
4	21.11.22	Architectural Patterns I
5	28.11.22	Architectural Patterns II
6	05.12.22	Antipatterns I
7	12.12.22	Antipatterns II
	19.12.22	No lecture
8	09.01.23	Testing Patterns I
9	16.01.23	Testing Patterns II
10	23.01.23	No lecture
11	30.01.23	Microservice Patterns
12	08.02.21	Course Review

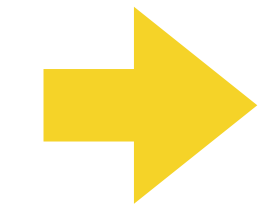


Roadmap of the lecture



- **Context and assumptions**
 - You have understood the basic concepts of patterns
 - You have implemented different design patterns, architectural patterns and refactored antipatterns
 - You have a good understanding about testing and experiences with the mock object pattern
- **Learning goals: at the end of this lecture you are able to**
 - Apply dependency injection with Guice and Spring
 - Test client server applications
 - Test code based on common test patterns
 - Apply test patterns to concrete situations
 - Differentiate between the testing patterns

Outline



Dependency injection

- Guice
- Spring
- Modern testing principles
- Test client server applications

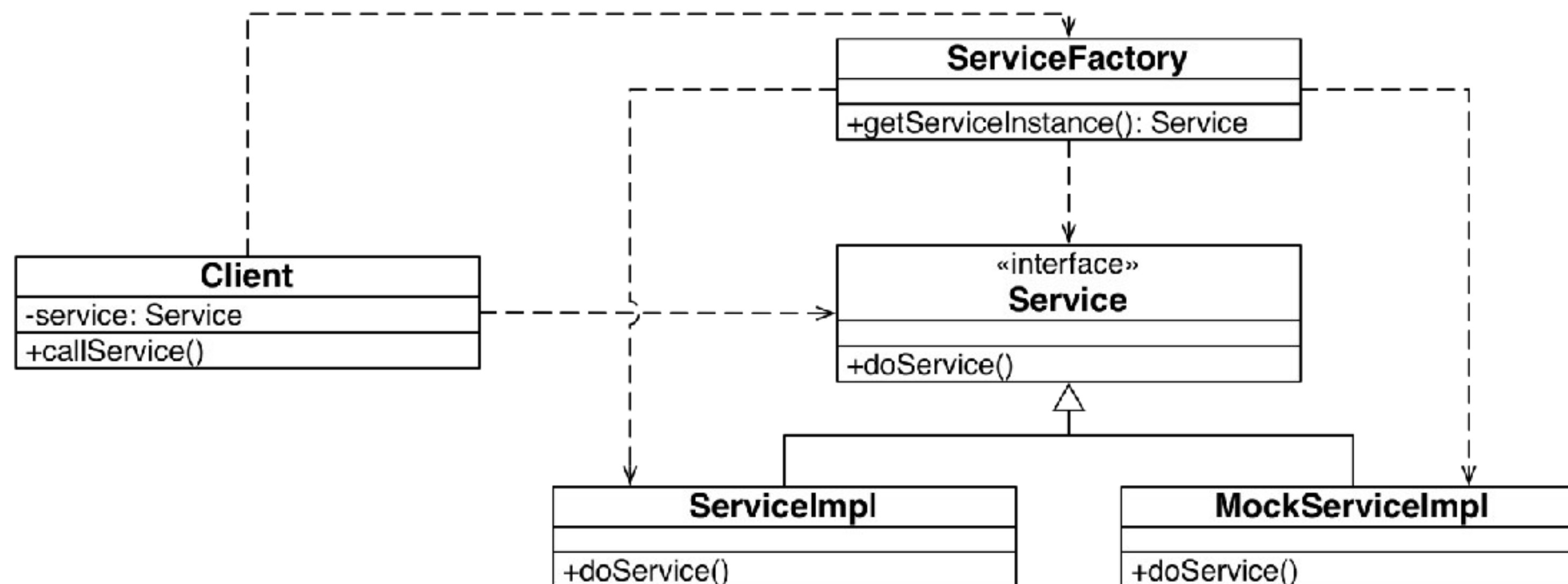
From state testing to behavior testing



- **Observation 1:** JUnit helps us to test the state of a SUT
 - **Limitation of Junit:** does not provide mechanisms to test a specific behavior, that is, which operations are called on collaborators and in which order
- **Observation 2:** Mock objects help to test behavior
 - **Limitation of mock objects:** create a high coupling between SUT and the rest of the system model (often not yet tested)
- **Goal:** reduce the high coupling as much as possible
 - Dependency injection

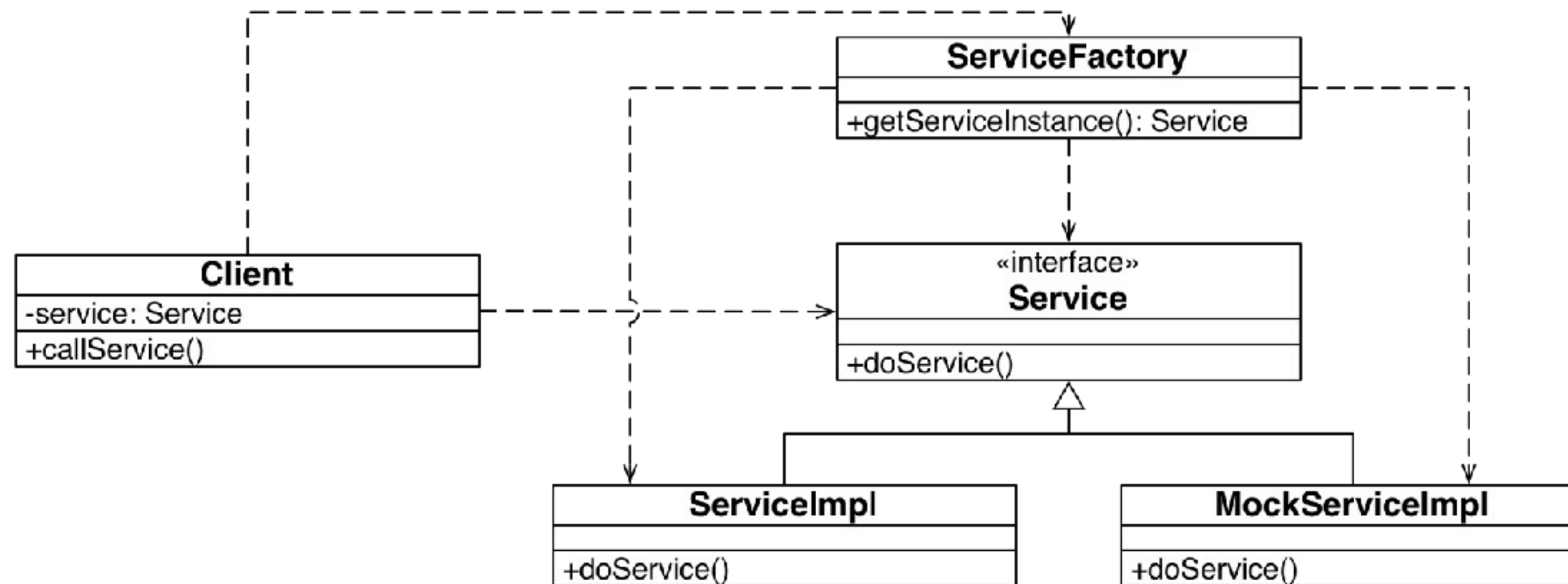
Instantiation with factories

- The use of the **factory design pattern** removes the dependency on the subclass implementation
- But now there is high coupling between **Client** and **ServiceFactory**
 - **ServiceFactory** provides access to the **ServiceImpl** and **MockServiceImpl**
 - **ServiceFactory** depends on these implementations.
 - As a result **Client** still has compile-time dependencies on the implementation classes



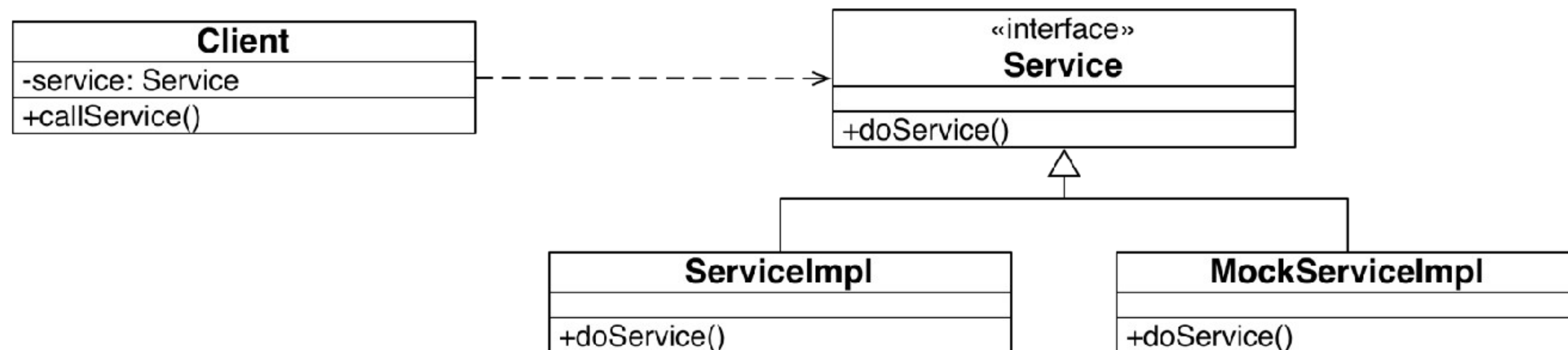
Instantiation with factories

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



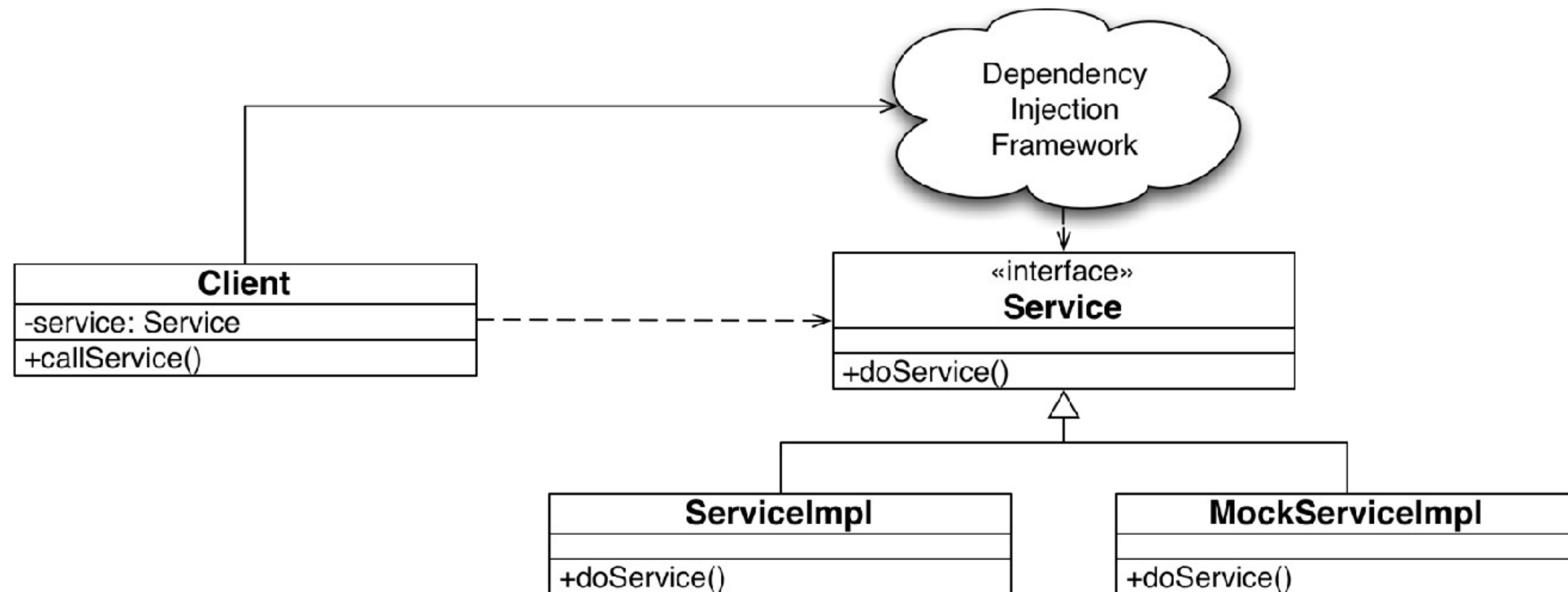
Introduction to dependency injection

- Client should only depend on the **Service** interface → **Low coupling!**
- **Needed:** mechanism that allows us to switch between different kinds of services easily



Introduction to dependency injection

- A **dependency injection framework** reduces the coupling for the **Client**
- **Client** now only knows **Service**
 - Binding (the selection of the correct subclass to be instantiated for a certain interface) is done by the **dependency injection framework**
 - There are no compile time dependencies between Client and implementation classes anymore
 - Because there is no factory, the **Client** class can be reused more easily



Dependency injection frameworks

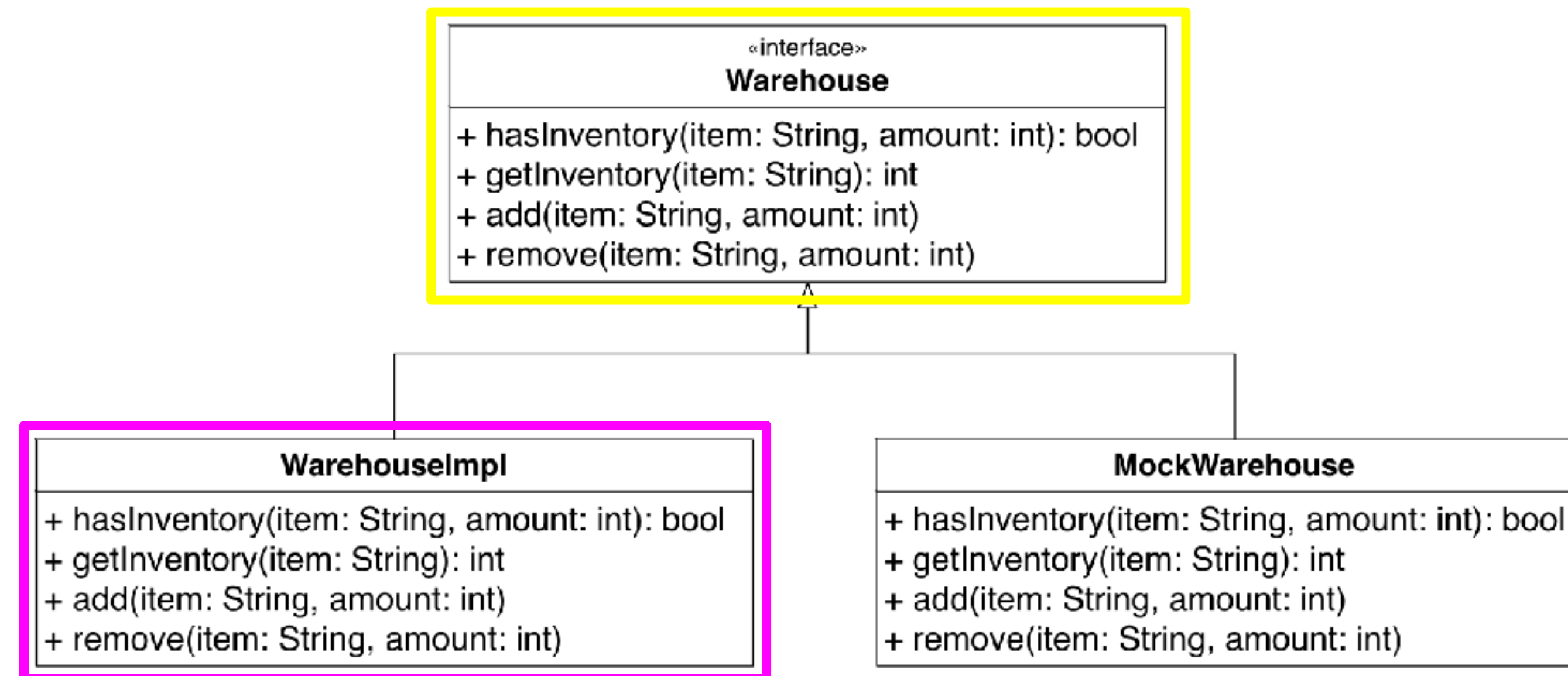
- There are many frameworks for dependency injection
- **Examples** for Java
 - ➔ **Guice:** uses Java annotations for configuration of the bindings
 - **Spring:** uses Java annotations or XML to configure the binding of the specific subclass implementation to the interface
- Dependency Injection is part of the Java standard
 - **JSR299** and **JSR330:** Java specification request #299 and #330
 - Contexts and dependency injection for Java EE (enterprise edition)
- Dependency injection for C++
 - Fruit (<https://github.com/google/fruit/wiki>)

- Guice (<http://github.com/google/guice>) uses the concept of a **Module** to bind a subclass implementation to an interface
 - It provides an abstract class **AbstractModule** with a protected method **configure()**
 - **AbstractModule** can have subclasses called Guice modules
 - Specification of **AbstractModule**
 - <https://google.github.io/guice/api-docs/latest/javadoc/com/google/inject/AbstractModule.html>
- Modules overwrite the **configure()** method to bind an implementation to a specification

Guice modules

- Two Guice modules are often used: **ProductionModule** and **TestModule**
- Here is [example](#) code for a **ProductionModule** that binds **WarehouseImpl** to **Warehouse**

```
public class ProductionModule extends AbstractModule {  
    public void configure() {  
        bind(Warehouse.class).to(WarehouseImpl.class)  
    }  
}
```



InventorySystem 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);
    }
}
```

InventorySystem with Guice

```
public class InventorySystem {
    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());
        InventorySystem inventorySystem = injector.getInstance(InventorySystem.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 InventorySystem {  
    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());  
        InventorySystem inventorySystem = injector.getInstance(InventorySystem.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!

An **injector** is created: it allows to specify bindings of implementations to interfaces → In this case it injects a **ProductionModule**

Two changes to the **Client**

4 steps to use Guice

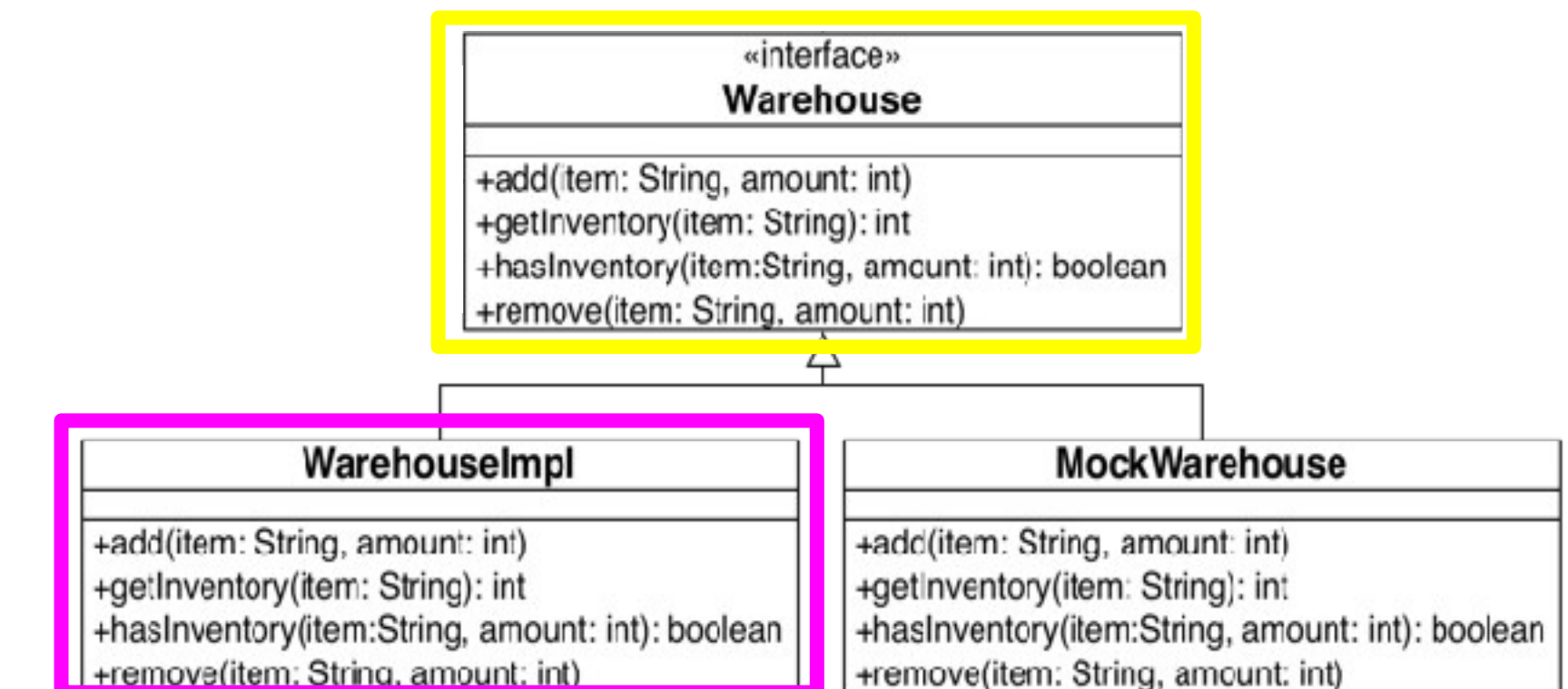
1. Tell Guice where to inject using the @Inject annotation:

- Constructor injection
- Method injection
- Field injection (with Java Annotation)

```
@Inject
private Warehouse warehouse;
```

2. Create a Module to define the Binding

```
@Override
protected void configure() {
    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: InventorySystem)

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


Guice and unit testing

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

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

- **@Inject** is used

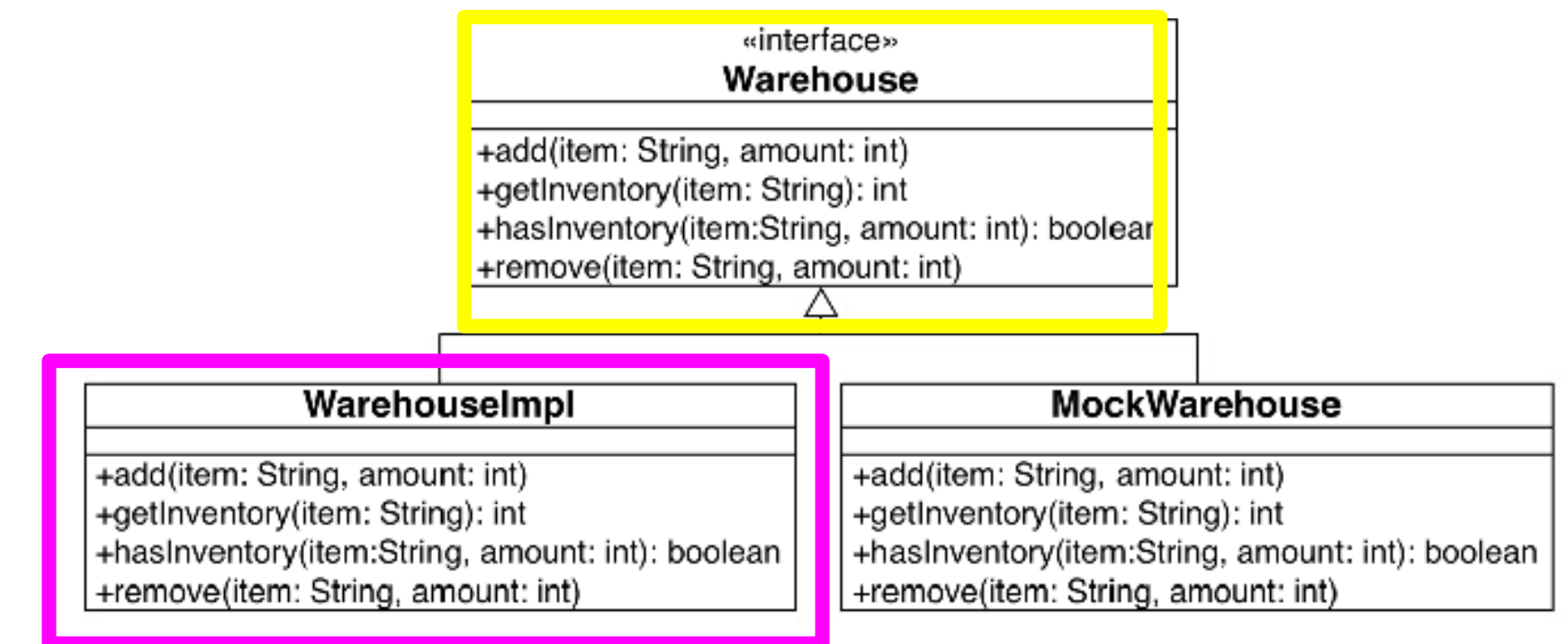
```
public class InventorySystem {  
    @Inject  
    private Warehouse warehouse;  
    ...  
}
```

- As a result you 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

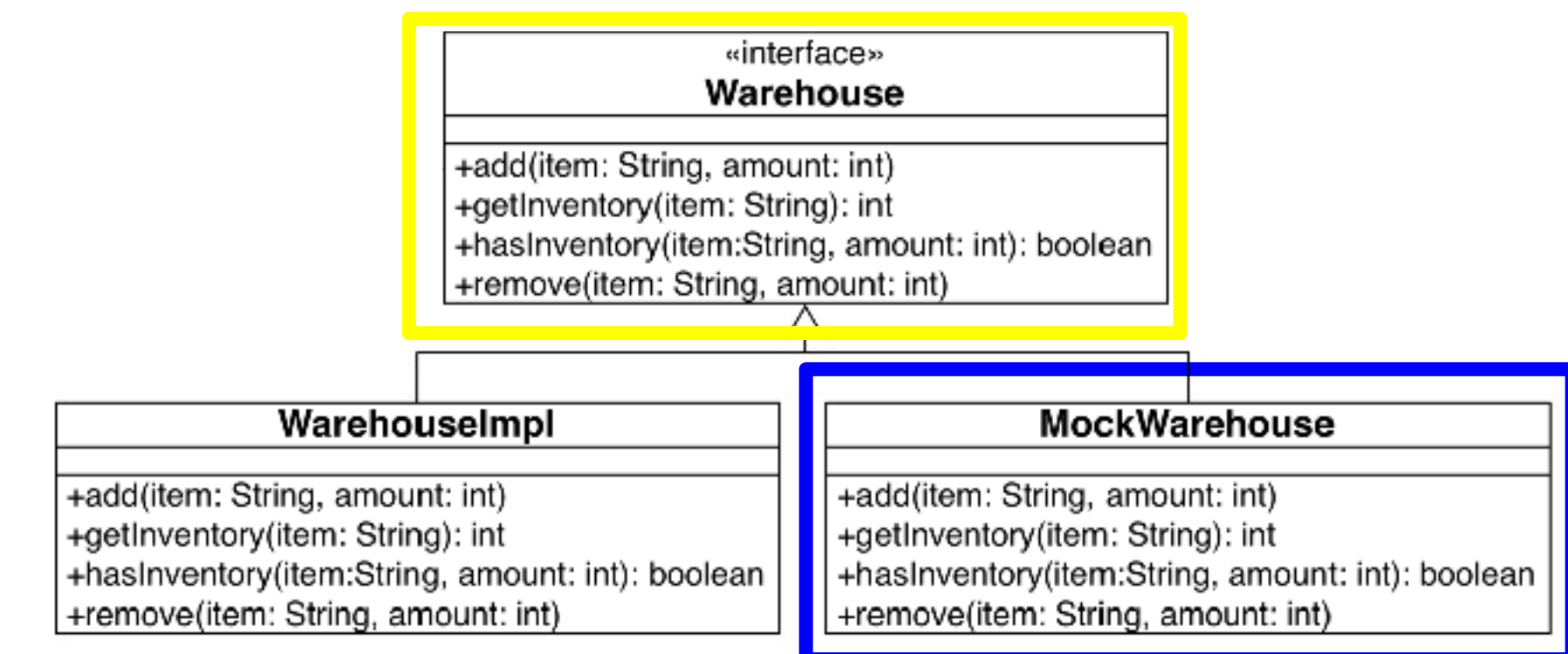
Guice module for the production code

```
public class ProductionModule extends AbstractModule {  
  
    @Override  
    protected void configure() {  
        bind(Warehouse.class).to(WarehouseImpl.class);  
    }  
}
```



Guice module for the unit test

```
public class TestModule extends AbstractModule {  
  
    @Override  
    protected void configure() {  
        Warehouse warehouseMock = EasyMock.createMock(Warehouse.class);  
        bind(Warehouse.class).toInstance(warehouseMock);  
    }  
}
```



Test of the **InventorySystem** using Guice and EasyMock



```
class InventorySystemTest {  
    private static String TALISKER = "Talisker";  
    private InventorySystem inventorySystem;  
    private Injector injector;  
  
    @BeforeEach  
    void setUp() throws Exception {  
        injector = Guice.createInjector(new TestModule());  
        inventorySystem = injector.getInstance(InventorySystem.class);  
    }  
  
    @Test  
    void addToWarehouse() {  
        Warehouse warehouseMock = injector.getInstance(Warehouse.class);  
        warehouseMock.add(TALISKER, 50);  
        expectLastCall().andVoid();  
        replay(warehouseMock);  
        inventorySystem.addToWarehouse(TALISKER, 50);  
        assertEquals(inventorySystem.getTotalStock(), 50);  
        verify(warehouseMock);  
    }  
}
```

Tell **Guice** to create an **injector** using **TestModule** that binds **Warehouse** to a **WarehouseMock**

Let **Guice** instantiate the **InventorySystem** and the **Warehouse** in it

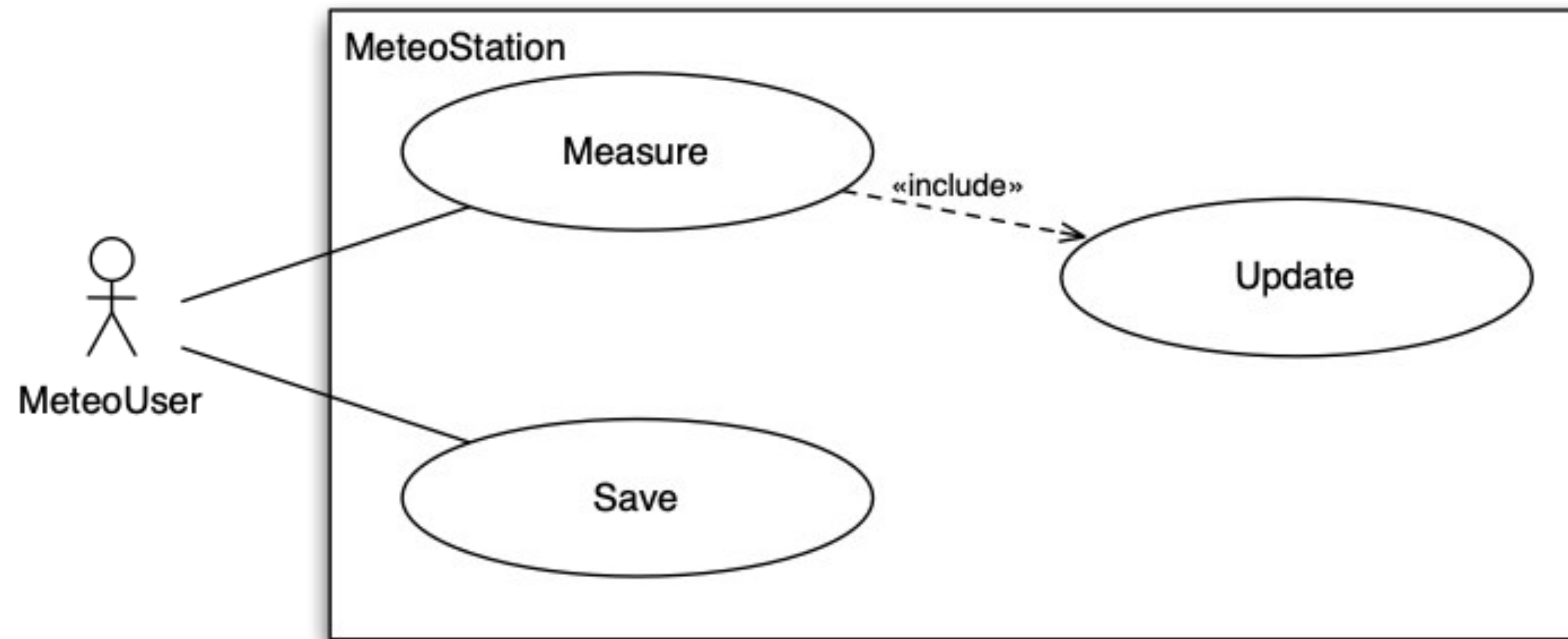
Retrieve the **WarehouseMock** from the **injector** to use it during the test

Specified behavior in the test case

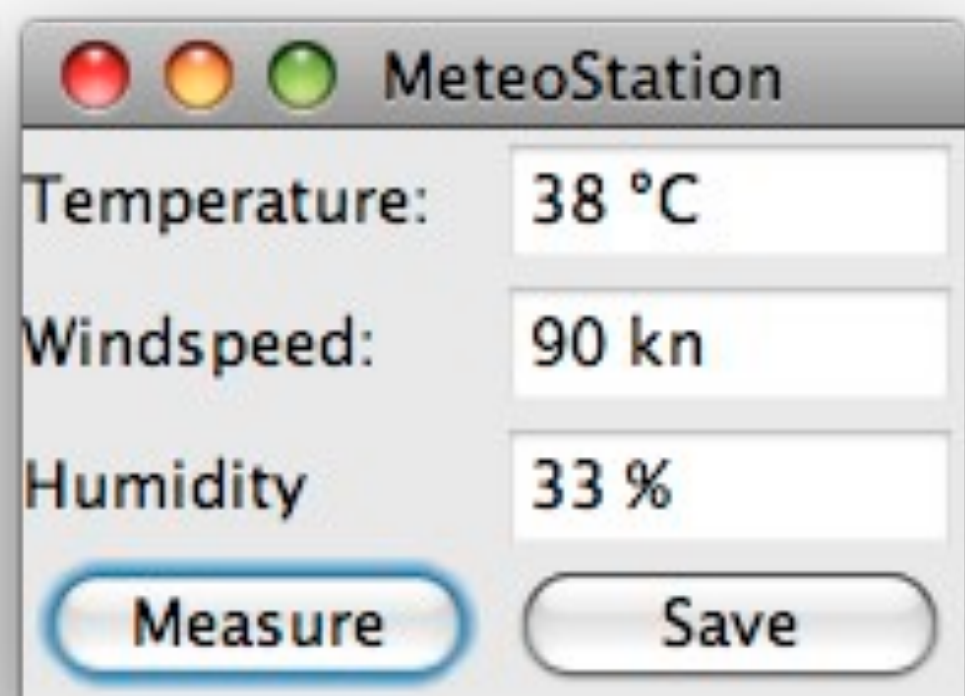
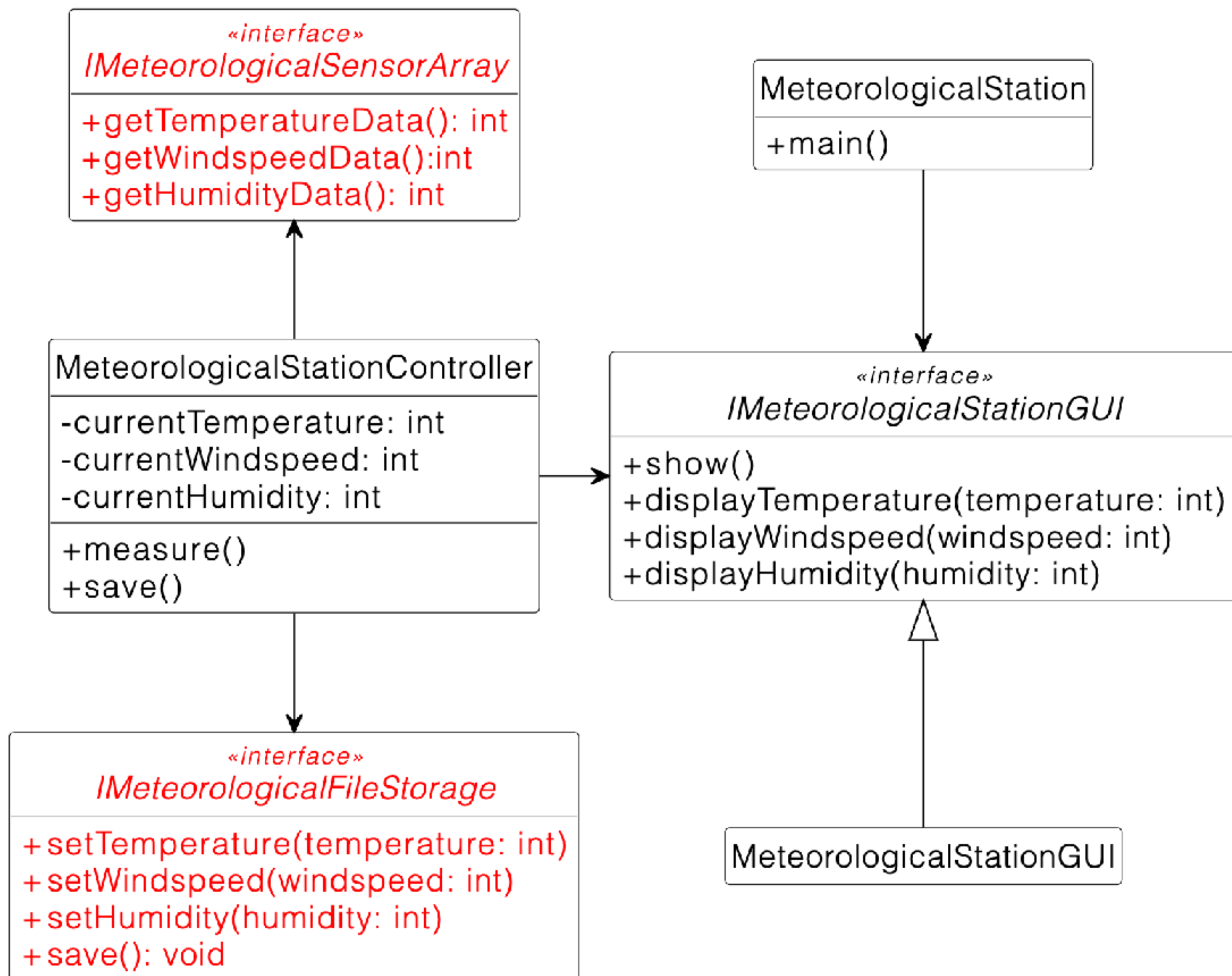
EasyMock validation

Problem statement for exercises

- The user MeteoUser can **measure** the temperature, wind speed and humidity values
- The application **updates** the current values when MeteoUser clicks the measure button
- The MeteoUser can **save** the current values to a text file

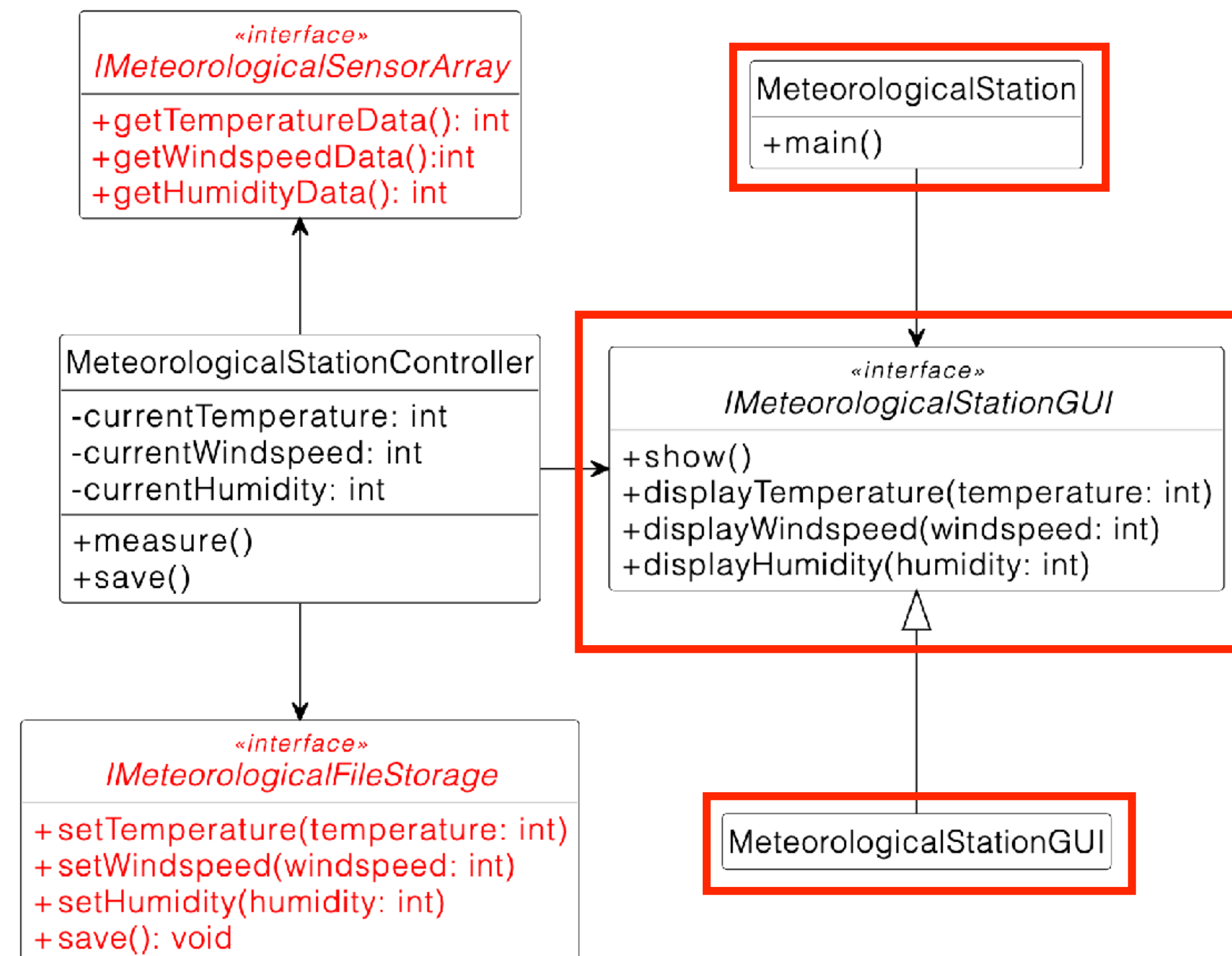


UML class diagram



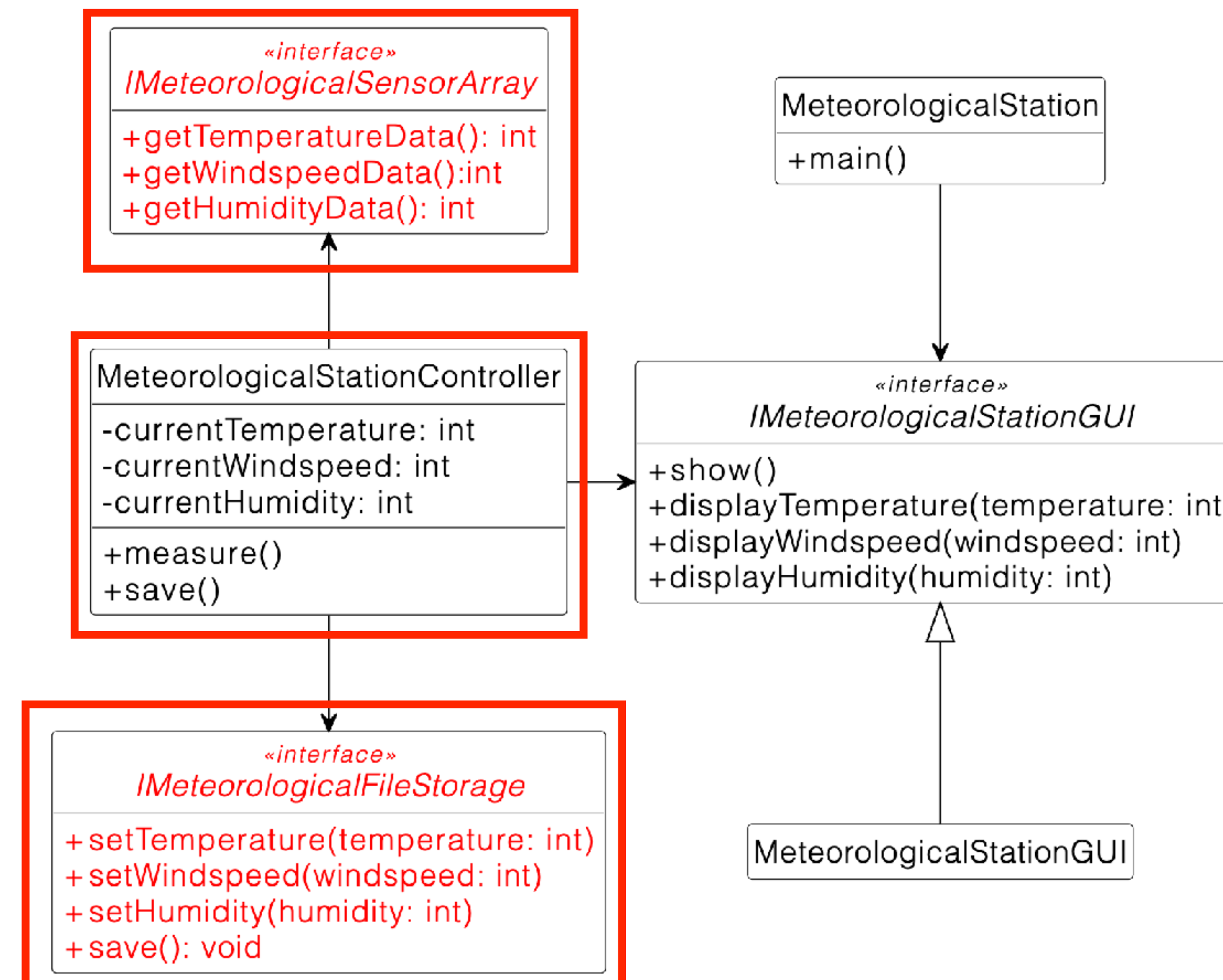
Meteorological station details

- **MeteorologicalStation** contains the main method and configures the application
- **IMeteorologicalStationGUI** declares an interface used to decouple the GUI from the MeteorologicalStationController. It provides methods that the controller can use to update the GUI
- **MeteorologicalStationGUI** is a Java Swing implementation of the application's GUI



Meteorological Station Details

- **MeteorologicalSensorArray** provides the meteorological data
- **MeteorologicalFileStorage** save the meteorological data to a text file
- **MeteorologicalStationController** mediates between the model (**MeteorologicalSensorArray**) and the view (**MeteorologicalStationGUI**)





L09E01 Dependency Injection with Guice

Not started.



Start exercise

Hard

Due date in 7 days



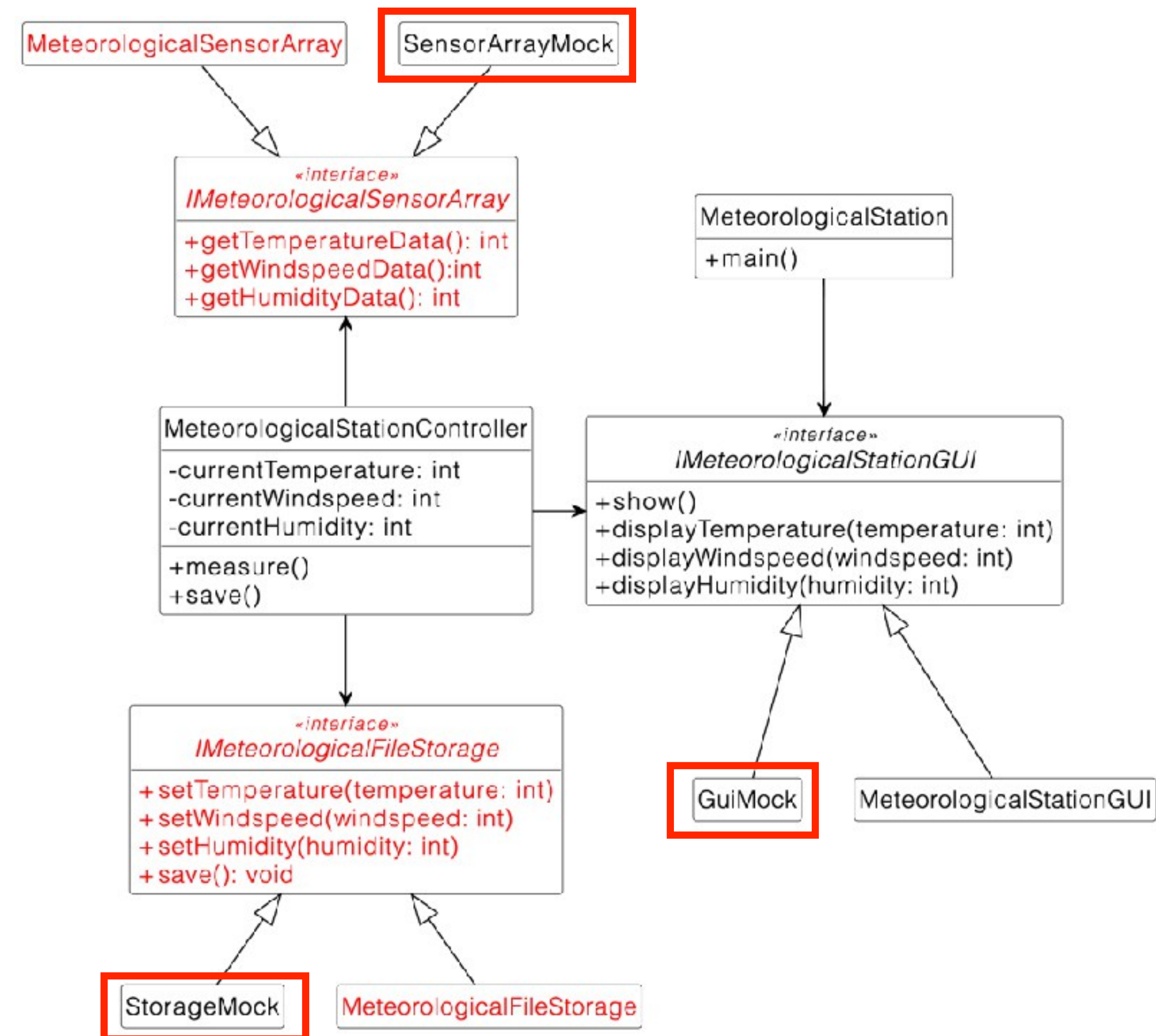
20 min



15 pts



- **Problem statement:** testing the meteorological station with **Guice** and dependency injection



- Prepare the **MeteorologicalStationController** for unit testing
- Remove compile time dependency between **MeteorologicalStationGUI** and **MeteorologicalStationController**
- Follow these steps to implement dependency injection using **Guice**:
 - 1. ProductionModule**
 - Create the **Guice ProductionModule** and configure the binding for **MeteorologicalStationGUI**
 - 2. MeteorologicalStation**
 - Instantiate an **Injector** with the **ProductionModule**
 - Instantiate **MeteorologicalStationGUI** using the injector (use a **IMeteorologicalStationGUI** property)
 - 3. MeteorologicalStationController**
 - Annotate the constructor with **@Inject** to point **Guice** where to inject the GUI
 - 4. MeteorologicalStationGUI**
 - Annotate the **MeteorologicalStationController** attribute with **@Inject**
 - Remove the line **controller = new MeteorologicalStationController(this)** in the constructor

Break



10 min

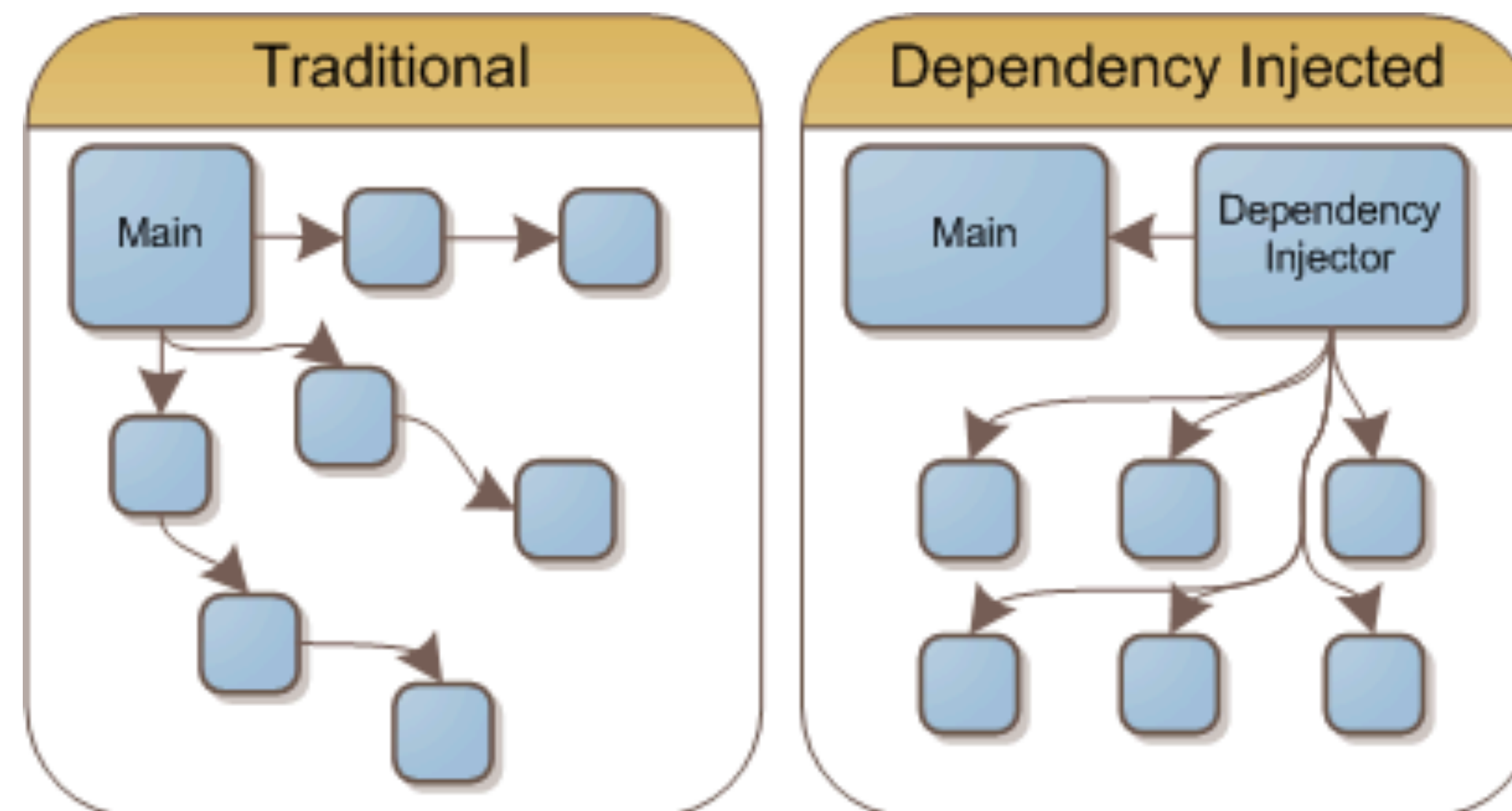
The lecture will continue at **11:15**

Outline

- Dependency injection
 - Guice
 - ➔ Spring
- Modern testing principles
- Test client server applications

Inversion of control

- Control of objects or portions of a program is transferred to a **container** or **framework**
- The framework takes **control of the flow of a program** and makes calls to the program's custom code
- To enable this, frameworks use **abstractions** with additional behavior built in
- If you want to add your own behavior, you need to **extend the classes** of the framework or plugin your own classes



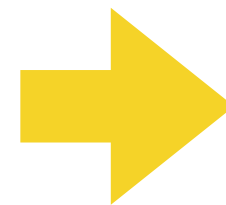
Inversion of control: **advantages**

- **Decoupling** the execution of a task from its implementation
- Making it **easier to switch** between different implementations
- **Greater modularity** of a program
- Greater **ease in testing** a program by isolating a component or mocking its dependencies and allowing components to communicate through contracts

Dependency injection

- Design principle to make
 - Your application easier to develop
 - Your code less coupled
 - Your code easier to test
- Objects get other required objects from outside

```
public class Store {  
    private Item item;  
  
    public Store() {  
        item = new ItemImpl();  
    }  
}
```



```
public class Store {  
    private Item item;  
  
    public Store(Item item) {  
        this.item = item;  
    }  
}
```

Spring

- Uses an application context for inversion of control
- 3 possibilities to use dependency injection
 1. **Constructor** injection
 2. **Method** injection (setter)
 3. **Field** injection (attribute)
- Autowiring dependencies
- Lazy instantiations



Application context



- Manages the objects of the application
- Uses dependency injection to achieve inversion of control (**IoC**)
- The interfaces **BeanFactory** and **ApplicationContext** represent the Spring **IoC** container
 - **BeanFactory** provides basic functionalities for managing beans
 - **ApplicationContext** is a sub interface of **BeanFactory** and provides more enterprise specific functionalities
 - Resolve messages
 - Support internationalization
 - Publish events
 - Provide application layer specific contexts

Spring bean

- An object that the Spring container instantiates, assembles, and manages
- Should you configure all the objects of our application as **Spring beans**?
 - As best practice, you should **not**
- You should define beans for
 - Service layer objects
 - Data access objects (DAOs, e.g. **Repository**)
 - Presentation objects (e.g. **RestController**)
 - Infrastructure objects such as Hibernate **SessionFactory**, Queue, etc
- You should **not** configure fine grained domain (entity) objects in the container
- DAOs and server layer objects create or load domain objects (e.g. from the database)

Bean definition in the configuration

Indicates that the class is a source of bean definitions

Used on a method to define a bean with a default name

```
@Configuration
public class AppConfig {

    @Bean
    public Item item() {
        return new ItemImpl();
    }

    @Bean
    public Store store() {
        return new Store(item());
    }
}
```

- For a bean with the default **singleton** scope, Spring first checks if a cached instance of the bean already exists and only creates a new one if it doesn't
- If we're using the **prototype** scope, the container returns a new bean instance for each method call

Constructor injection

- In Spring, no annotations are needed

```
private final Item item;  
  
public Store(Item item) {  
    this.item = item;  
}
```

Method injection

- The container will call setter methods of our class, after instantiating the object (possibly with the default constructor with no arguments)

This annotation tells Spring to invoke this method with a bean object

```
@Autowired  
public void setStore(Store store) {  
    this.store = store;  
}
```

Field injection

- Inject dependencies by marking them with an **@Autowired** annotation

This annotation tells Spring to inject a bean object

```
public class Store {  
    @Autowired  
    private Item item;  
}
```

Which option should be preferred: **constructor injection**



- Leads to more robust code
- All required dependencies are available at initialization time
- Allows to identifying code smells
 - **Example:** large number of arguments → **problem:** too many responsibilities
- Simplifies writing unit tests
- Prevents **NullPointerExceptions** and other errors
- **Immutability:** once the framework creates a bean, it cannot alter its dependencies anymore
 - Easier to debug
 - Easier to use in multi-threaded environments

- Allows the Spring container to automatically resolve dependencies between collaborating beans by inspecting the beans that have been defined
- There are 4 modes of **autowiring** a bean
 1. **no**: the default value – this means no **autowiring** is used for the bean and we have to explicitly name the dependencies
 2. **byName**: **autowiring** is done based on the name of the property, therefore Spring will look for a bean with the same name as the property that needs to be set
 3. **byType**: similar to the **byName autowiring**, only based on the type of the property: this means Spring will look for a bean with the same type of the property to set (if there's more than one bean of that type, the framework throws an exception)
 4. **constructor**: **autowiring** is done based on constructor arguments, meaning Spring will look for beans with the same type as the constructor arguments

Autowiring examples

```
@Bean(autowire = Autowire.BY_TYPE)
public class Store {

    private Item item;

    public setItem(Item item){
        this.item = item;
    }
}
```

```
public class Store {

    @Autowired
    private Item item;
}
```

```
public class Store {

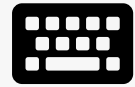
    @Autowired
    @Qualifier("item1")
    private Item item;
}
```

Reference by name



L09E02 Dependency Injection with Spring

Not started.



Start exercise

Medium

Due date in 7 days



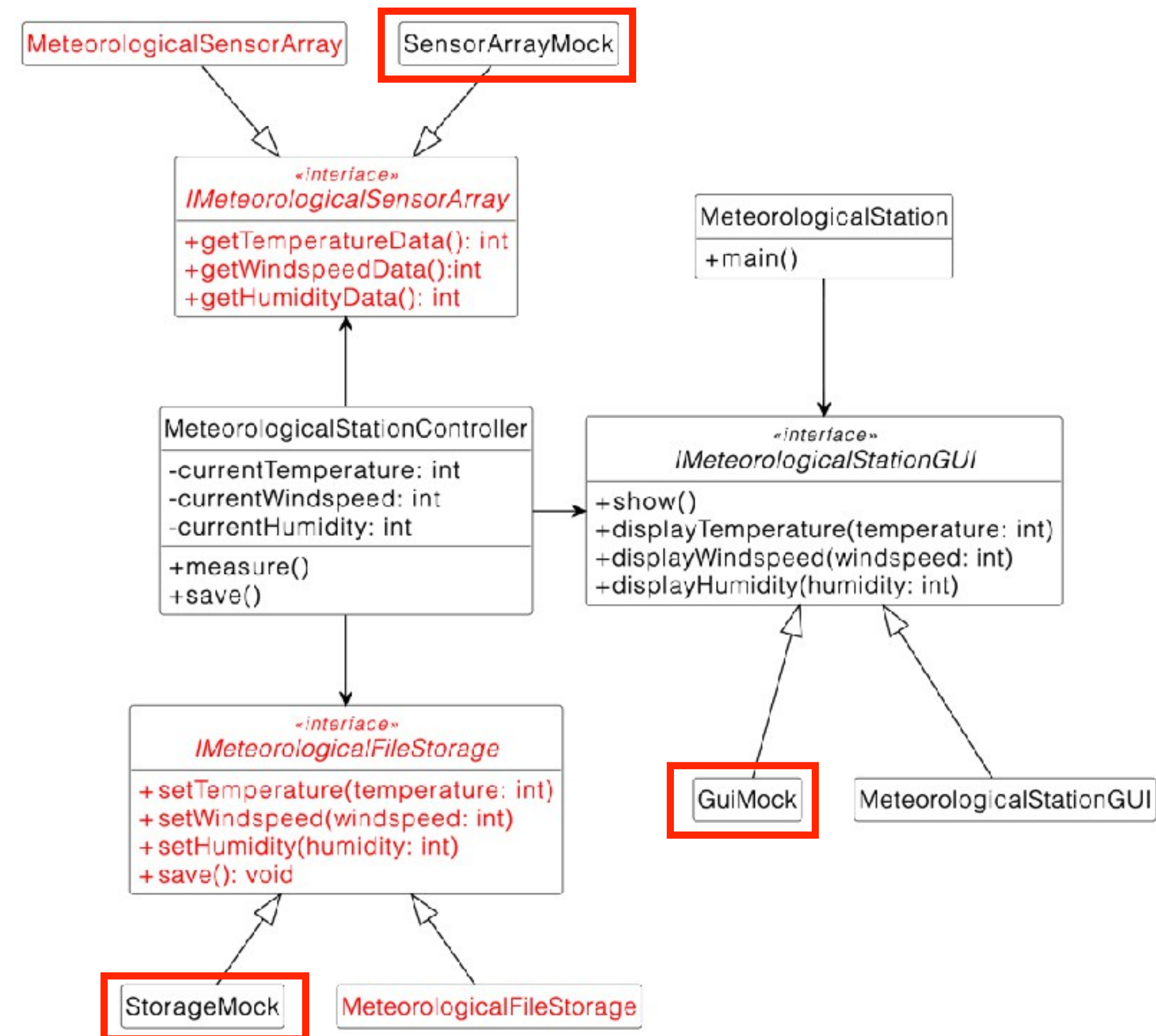
20 min



12 pts



- **Problem statement:** testing the meteorological station with Spring and dependency injection



Outline

- Dependency injection
 - Guice
 - Spring
- ➔ Modern testing principles
 - Test client server applications

- [AssertJ](#) is an powerful and mature assertion library with a type safe API, a variety of assertions and descriptive failure messages
- There is an assertion for everything you want to assert
- Prevents writing complex assertion logic with loops and conditions and keeps tests short

```
assertThat(actualProduct).isEqualToIgnoringGivenFields(expectedProduct, "id");

assertThat(actualProductList).containsExactly(createProductDT0("1", "Smartphone", 250.00),
        createProductDT0("1", "Smartphone", 250.00));

assertThat(actualProductList).usingElementComparatorIgnoringFields("id")
        .containsExactly(expectedProduct1, expectedProduct2);

assertThat(actualProductList).extracting(Product::getId).containsExactly("1", "2");

assertThat(actualProductList)
        .anySatisfy(product -> assertThat(product.getDateCreated()).isBetween(instant1, instant2));

assertThat(actualProductList).filteredOn(product -> product.getCategory().equals("Smartphone"))
        .allSatisfy(product -> assertThat(product.isLiked()).isTrue());
```

Modern testing principles

➔ General

- Write small and specific tests
- Self contained tests
- Dumb tests are great: compare the output with hard coded values
- Test close to the reality
- Make the implementation testable
- Assertions in Java
- Use Junit5 features
- Mock remote services

Source: <https://phauer.com/2019/modern-best-practices-testing-java>

Modern testing principles (1)

- General

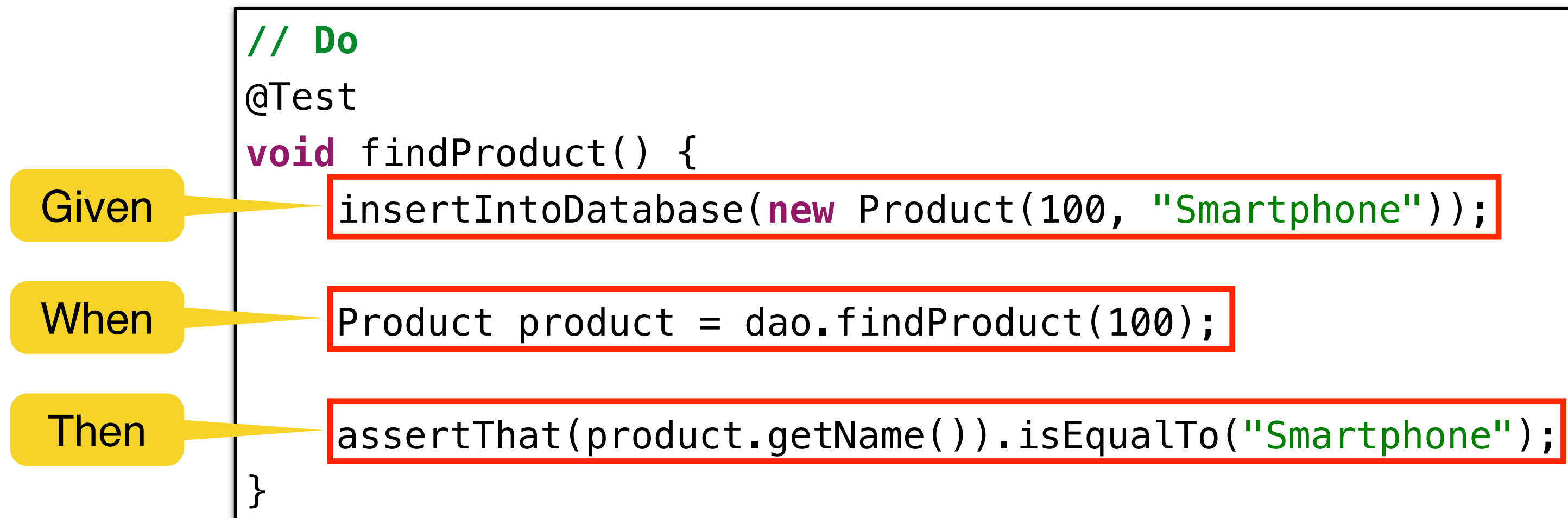
➔ Given, when, then

- Use prefixes **actual*** and **expected***
 - Use fixed data instead of randomized data
- Write small and specific tests
 - Use helper functions
 - Don't overuse variables
 - Don't extend existing tests to "just test one more tiny thing"
 - Assert only what you want to test
- Self contained tests
 - Don't hide the relevant parameters (in helper functions)
 - Insert test data right in the test method
 - Favor composition over inheritance

Source: <https://phauer.com/2019/modern-best-practices-testing-java>

Given, when, then

- A test should contain 3 blocks which are separated by one empty line
 1. **Given (Input):** test preparation like creating data or configure mocks (precondition)
 2. **When (Action):** Call the method or action that you like to test
 3. **Then (Output):** Execute assertions to verify the correct output or behavior of the action (postcondition)
- Each block of code should be as short as possible
- Invoke other methods to shorten these blocks



```
// Do
@Test
void findProduct() {
    insertIntoDatabase(new Product(100, "Smartphone"));
    Product product = dao.findProduct(100);
    assertThat(product.getName()).isEqualTo("Smartphone");
}
```

The diagram illustrates the mapping of the Given-When-Then pattern to a JUnit test method. Three yellow callout boxes on the left are connected by arrows to specific lines of code in the method `findProduct()`:

- Given** points to the line `insertIntoDatabase(new Product(100, "Smartphone"));`
- When** points to the line `Product product = dao.findProduct(100);`
- Then** points to the line `assertThat(product.getName()).isEqualTo("Smartphone");`

Use prefixes **actual*** and **expected***

// Don't

```
ProductDTO product1 = requestProduct(1);
```

```
ProductDTO product2 = new ProductDTO("1", List.of(State.ACTIVE, State.REJECTED))  
assertThat(product1).isEqualTo(product2);
```

// Do

```
ProductDTO actualProduct = requestProduct(1);
```

```
ProductDTO expectedProduct = new ProductDTO("1", List.of(State.ACTIVE, State.REJECTED))  
assertThat(actualProduct).isEqualTo(expectedProduct);
```

Use fixed data instead of randomized data

- Avoid randomized data as it can lead to flaky tests which can be hard to debug

```
// Don't
Instant ts1 = Instant.now(); // 1557582788
Instant ts2 = ts1.plusSeconds(1); // 1557582789
int randomAmount = new Random().nextInt(500); // 232
UUID uuid = UUID.randomUUID(); // d5d1f61b-0a8b-42be-b05a-bd458bb563ad
```

- Instead, use fixed values for everything
- They will create reproducible tests, which are easy to debug

```
// Do
Instant ts1 = Instant.ofEpochSecond(1550000001);
Instant ts2 = Instant.ofEpochSecond(1550000002);
int amount = 50;
UUID uuid = UUID.fromString("00000000-000-0000-0000-000000000001");
```

Modern testing principles (1)

- General
 - Given, when, then
 - Use prefixes **actual*** and **expected***
 - Use fixed data instead of randomized data
- Write small and specific tests
 - Use helper functions
 - Don't overuse variables
 - ➔ Don't extend existing tests to "just test one more tiny thing"
 - Assert only what you want to test
- Self contained tests
 - Don't hide the relevant parameters (in helper functions)
 - Insert test data right in the test method
 - Favor composition over inheritance

Source: <https://phauer.com/2019/modern-best-practices-testing-java>

Don't extend existing tests to “just test one more tiny thing”

- Small tests are important so that you can quickly find the issue in case one test fails

```
// Don't
class ProductControllerTest {
    @Test
    public void happyPath() {
        // a lot of code comes here...
    }
}
```

```
// Do
class ProductControllerTest {
    @Test
    public void multipleProductsAreReturned() {}
    @Test
    public void allProductValuesAreReturned() {}
    @Test
    public void filterByCategory() {}
    @Test
    public void filterByDateCreated() {}
}
```


Modern testing principles (1)

- General
 - Given, when, then
 - Use prefixes **actual*** and **expected***
 - Use fixed data instead of randomized data
- Write small and specific tests
 - Use helper functions
 - Don't overuse variables
 - Don't extend existing tests to "just test one more tiny thing"
 - Assert only what you want to test
- Self contained tests
 - ➔ Don't hide the relevant parameters (in helper functions)
 - Insert test data right in the test method
 - Favor composition over inheritance

Source: <https://phauer.com/2019/modern-best-practices-testing-java>

Don't hide the relevant parameters (in helper functions)



```
// Don't  
insertIntoDatabase(createProduct());  
List<ProductDTO> actualProducts = requestProductsByCategory();  
assertThat(actualProducts).containsOnly(new ProductDTO("1", "Office"));
```

```
// Do  
insertIntoDatabase(createProduct("1", "Office"));  
List<ProductDTO> actualProducts = requestProductsByCategory("Office");  
assertThat(actualProducts).containsOnly(new ProductDTO("1", "Office"));
```

Favor composition over inheritance

```
// Don't  
class SimpleBaseTest {}  
class AdvancedBaseTest extends SimpleBaseTest {}  
class AllInklusiveBaseTest extends AdvancedBaseTest {}  
class MyTest extends AllInklusiveBaseTest {}
```

- Those hierarchies are hard to understand and you likely end up extending a base test that contains a lot of stuff that the current test doesn't need
- “Prefer duplication over the wrong abstraction” (Sandi Metz)
- Instead use composition and delegate important setup and teardown tasks into other classes that can be invoked in all tests

Modern testing principles (2)

- Dumb tests are great: compare the output with hard coded values
 - Don't reuse production code
 - ➔ Don't rewrite production logic
 - Don't write too much logic
- Test close to the reality
 - Focus on testing a complete functional requirement (vertical slide)
 - Don't use in memory databases for tests
- Make the implementation testable
 - Don't use static access
 - Parameterize
 - Use constructor injection
 - Don't use **Instant.now()** or **new Date()**
 - Separate asynchronous execution and actual logic

Source: <https://phauer.com/2019/modern-best-practices-testing-java>

Don't rewrite production code

- Mapping code is a common example where the logic in tests is rewritten
- You'll most likely end up rewriting the production logic in the test code, which can contains bugs

```
// Don't
ProductEntity inputEntity = new ProductEntity(1, "evelope", "office", false, true, 200, 10.0);
insertIntoDatabase(input);

ProductDTO actualDTO = requestProduct(1);

// mapEntityToDto() contains the same mapping logic as the production code
ProductDTO expectedDTO = mapEntityToDto(inputEntity);
assertThat(actualDTO).isEqualTo(expectedDTO);
```

- Instead, compare the actualDTO with a manually created reference object with hard-coded values
- That's simple, easy to understand and less error-prone

```
// Do
ProductDTO expectedDTO = new ProductDTO("1", "evelope", new Category("office"), List.of(States.ACTIVE, States.REJECTED))
assertThat(actualDTO).isEqualTo(expectedDTO);
```

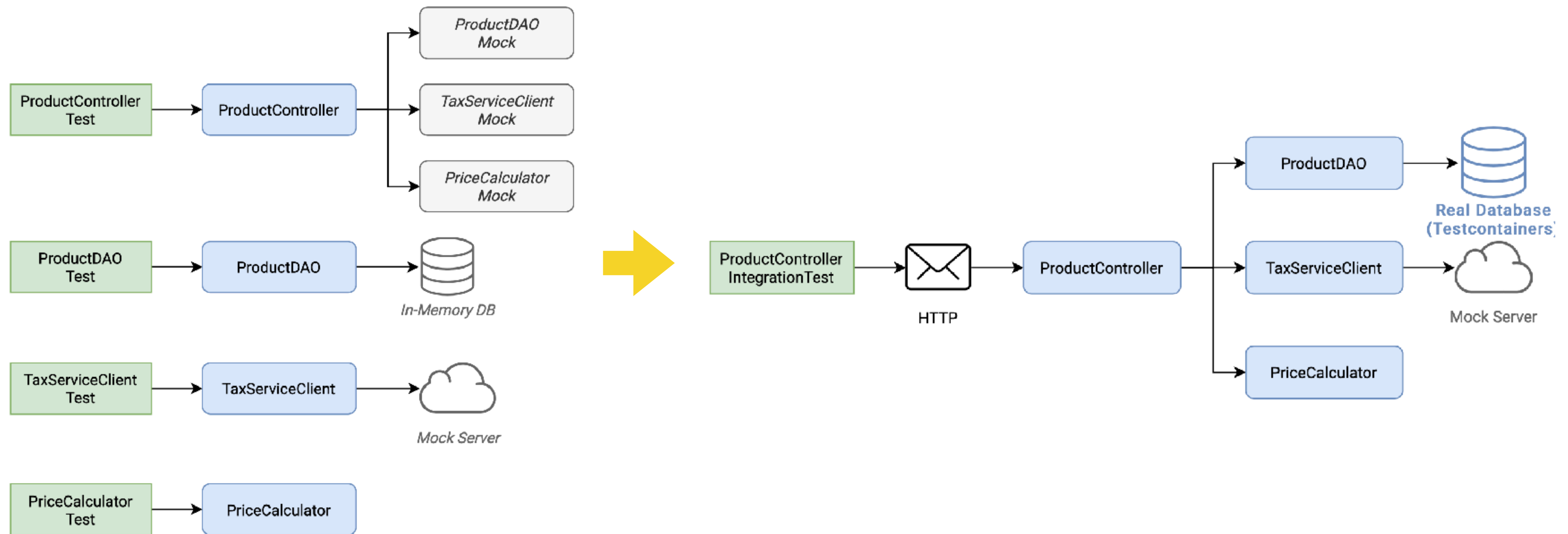
Modern testing principles (2)

- Dumb tests are great: compare the output with hard coded values
 - Don't reuse production code
 - Don't rewrite production logic
 - Don't write too much logic
- Test close to the reality
 - ➔ Focus on testing a complete functional requirement (vertical slice)
 - Don't use in memory databases for tests
- Make the implementation testable
 - Don't use static access
 - Parameterize
 - Use constructor injection
 - Don't use **Instant.now()** or **new Date()**
 - Separate asynchronous execution and actual logic

Source: <https://phauer.com/2019/modern-best-practices-testing-java>

Focus on testing a complete functional requirement

- Unit testing each class in isolation and with mocks comes with drawbacks
- Focus on integration test (= wiring real objects together and test all at once)



Modern testing principles (2)

- Dumb tests are great: compare the output with hard coded values
 - Don't reuse production code
 - Don't rewrite production logic
 - Don't write too much logic
- Test close to the reality
 - Focus on testing a complete functional requirement (vertical slice)
 - Don't use in memory databases for tests
- Make the implementation testable
 - Don't use static access
 - Parameterize
 - ➔ Use constructor injection
 - Don't use **Instant.now()** or **new Date()**
 - Separate asynchronous execution and actual logic

Source: <https://phauer.com/2019/modern-best-practices-testing-java>

Use constructor injection

- Field injection would lead to poor testability: you have to bootstrap the DI environment in your tests or do reflection
- Constructor injection is the preferred way because it allows you to easily control the dependent object in the test

```
// Do
public class ProductController {

    private ProductDAO dao;
    private TaxClient client;

    public CustomerResource(ProductDAO dao, TaxClient client) {
        this.dao = dao;
        this.client = client;
    }
}
```

Don't use `Instant.now()` or `new Date()`

- Don't get the current timestamp by calling **`Instant.now()`** or **`new Date()`** in your production code
- The created timestamp **cannot be controlled** by the test, because it's always different in every test execution
- Instead, use Java's **`Clock`** class
- You can create a mock for the clock in tests, pass it to **`ProductDAO`** and configure the **clock mock** to return a **fixed timestamp**

```
// Don't
public class ProductDAO {
    public void updateProductState(String productId) {
        Instant now = Instant.now();
        //...
    }
}
```

```
// Do
public class ProductDAO {
    private Clock clock;

    public ProductDAO(Clock clock) {
        this.clock = clock;
    }

    public void updateProductState(String productId) {
        Instant now = clock.instant();
        // ...
    }
}
```

Modern testing principles (3)

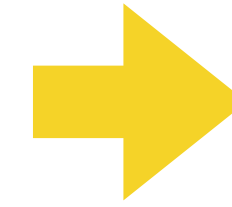
- Assertions in Java
 - ✓ Use AssertJ
 - ➔ Avoid **assertTrue()** and **assertFalse()**
 - Use Awaitility for asserting asynchronous code
- Use Junit5 features
 - Use parameterized tests
 - Group the tests
 - Readable test names with **@DisplayName**
- Mock remote services

Source: <https://phauer.com/2019/modern-best-practices-testing-java>

Avoid `assertTrue()` and `assertFalse()`

// Don't

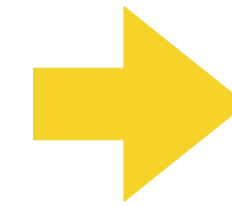
```
assertTrue(actualProductList.contains(expectedProduct));  
assertTrue(actualProductList.size() == 5);  
assertTrue(actualProduct instanceof Product);
```



expected: <true> but was: <false>

// Do

```
assertThat(actualProductList).contains(expectedProduct);  
assertThat(actualProductList).hasSize(5);  
assertThat(actualProduct).isInstanceOf(Product.class);
```



Expecting:

<[Product[id=1, name='Samsung Galaxy']]>

to contain:

<[Product[id=2, name='iPhone']]>

but could not find:

<[Product[id=2, name='iPhone']]>

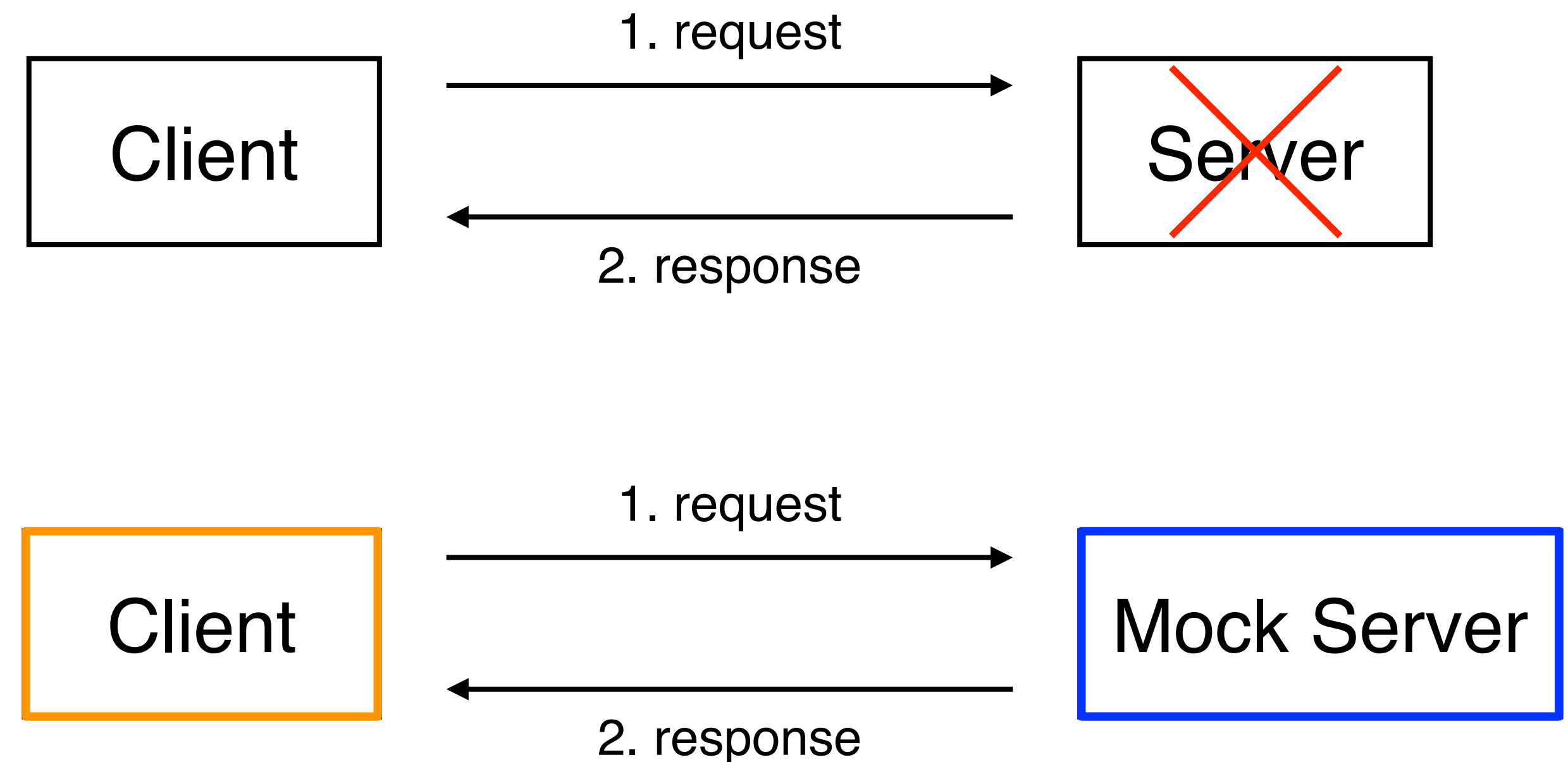
Using AssertJ: <https://assertj.github.io/doc>





Outline

- Dependency injection
 - Guice
 - Spring
 - Modern testing principles
- ➔ Test client server applications

Test **clients** by mocking remote services

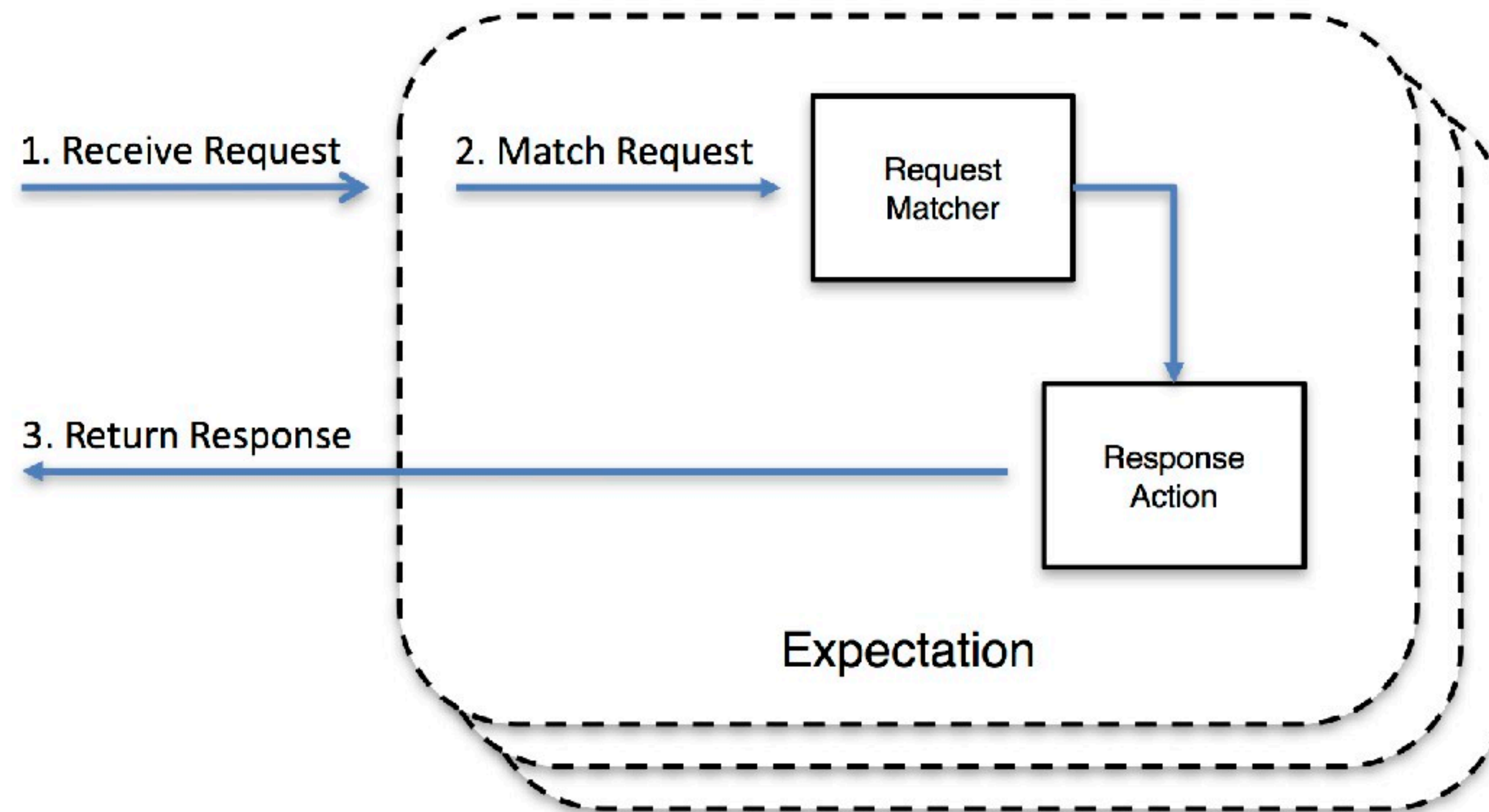
- In order to write integration **client** tests, you need to mock the remote service
- There are different frameworks which help you to mock remote services
 - [OkHttp WebMockServer](#)
 - [WireMock](#)
 - [Testcontainer Mockserver](#)
 - [Spring MockRestServiceServer](#)



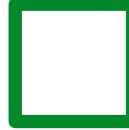


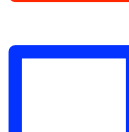
-  System model
-  System under test (SUT)
-  Test model
-  Collaborators

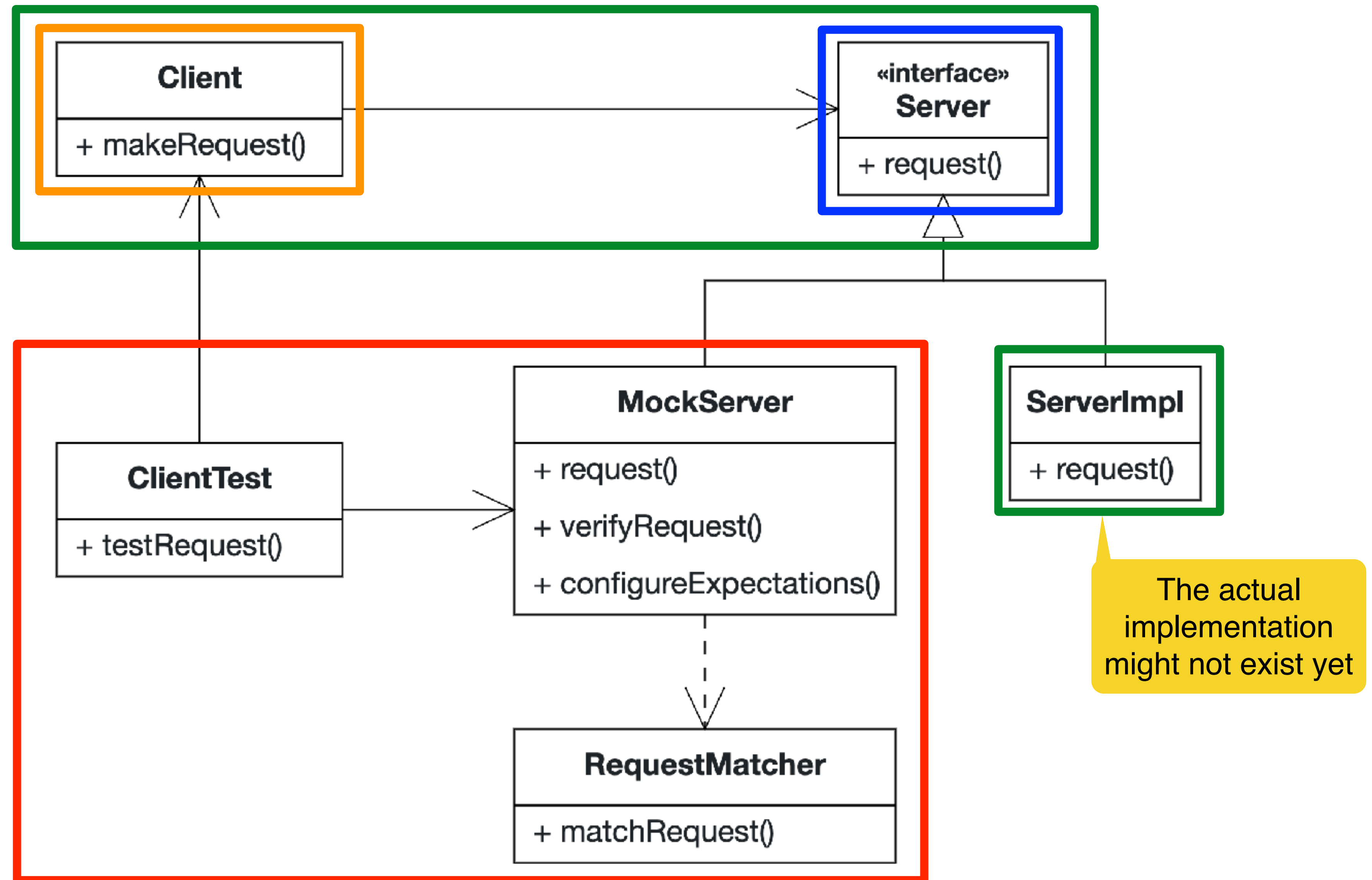
Test **clients** by mocking remote services

- In order to write integration **client** tests, you need to mock the remote service

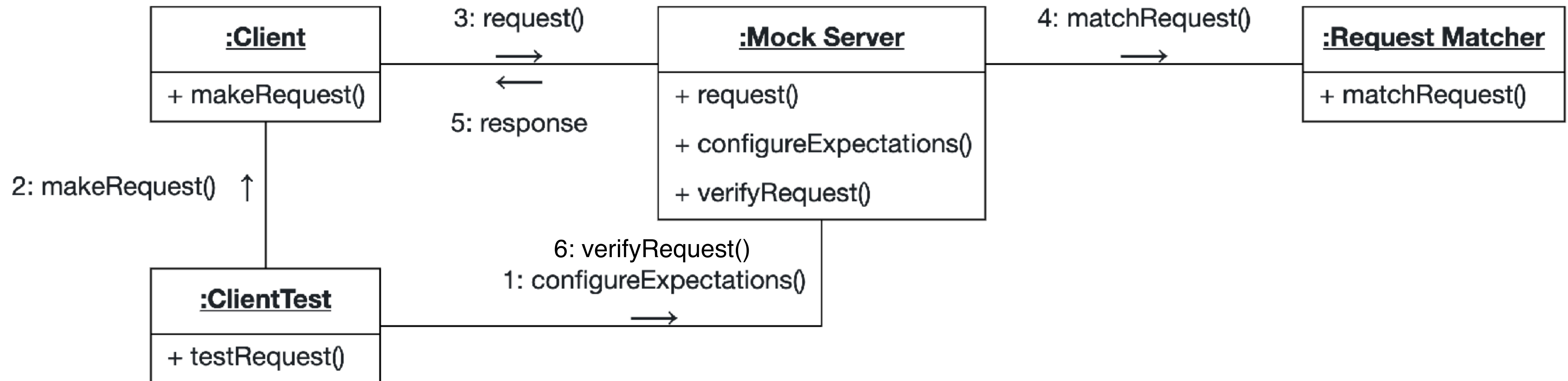


Structure of the client test

-  System model
-  System under test (SUT)
-  Test model
-  Collaborators

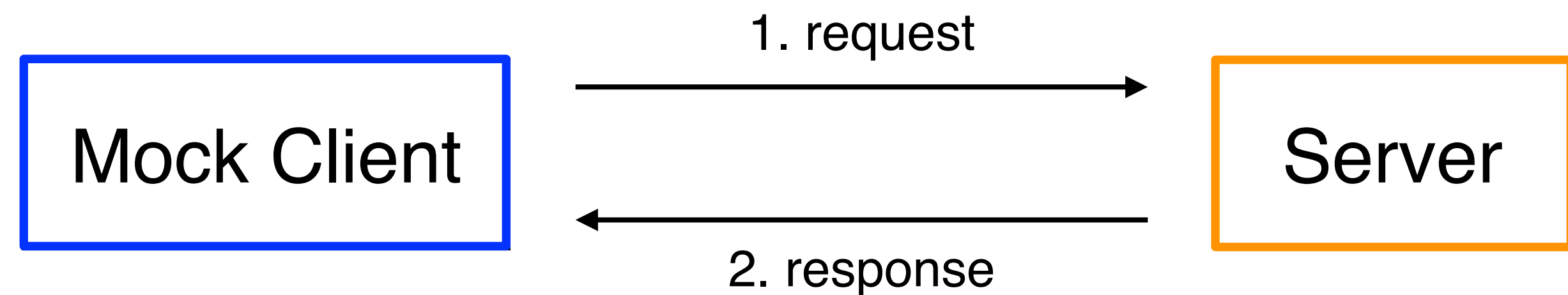
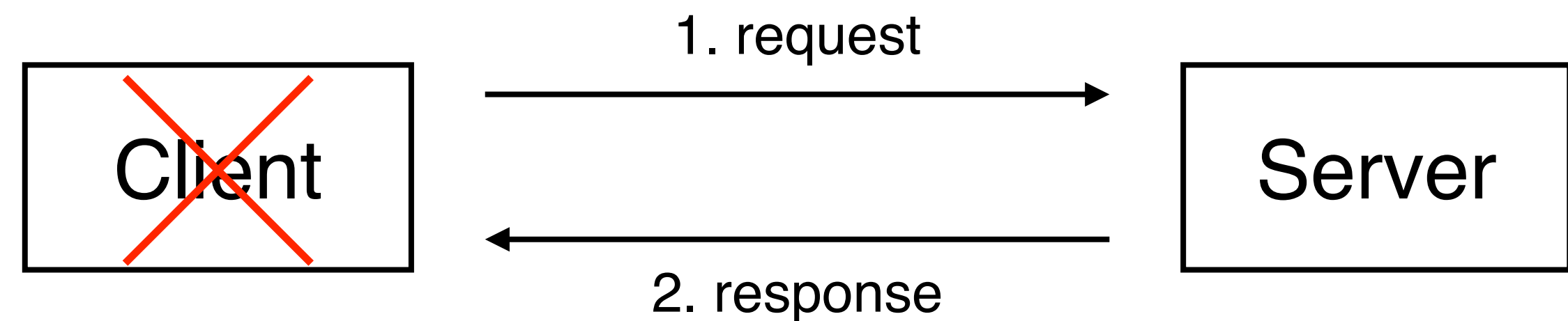


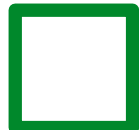



Dynamic behavior of the client test



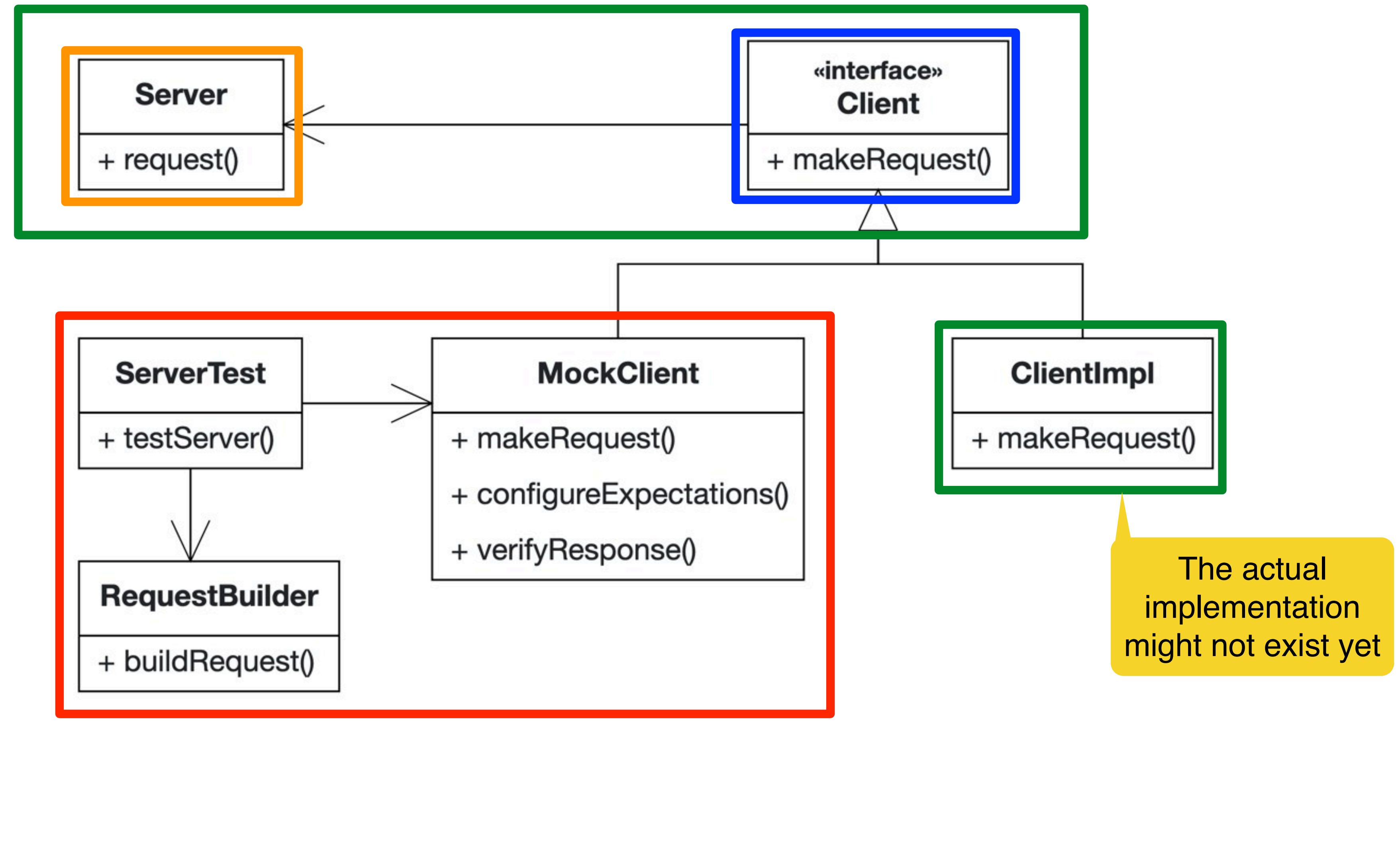
Test **servers** by mocking client calls

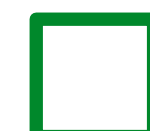



- In order to write integration server test, you need to mock the client
- In particular invoking REST calls and verifying the correct responses is necessary
- **Example:** Spring [MockMvc](#)



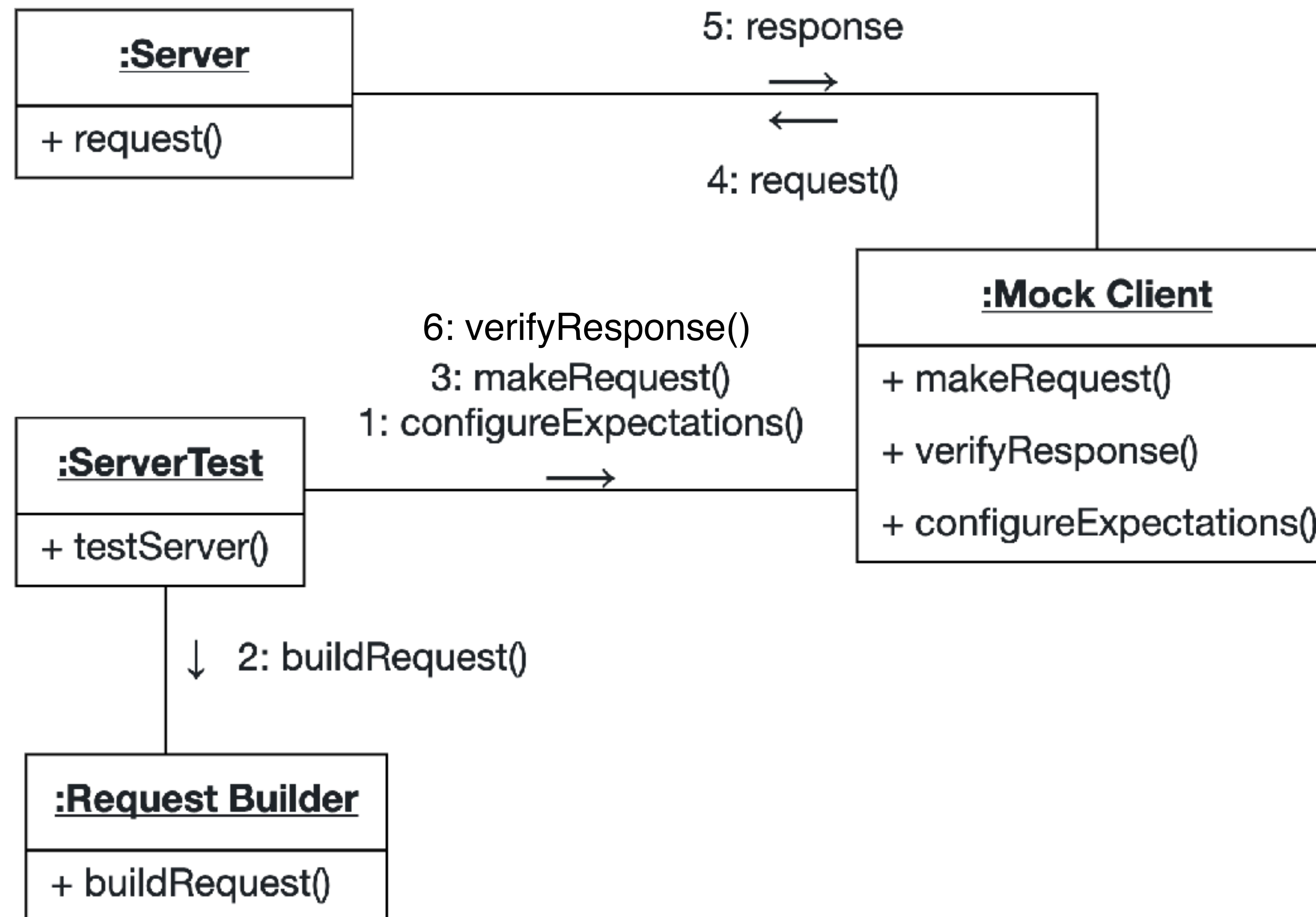
-  System model
-  System under test (SUT)
-  Test model
-  Collaborators

Structure of the server tests



-  System model
-  System under test (SUT)
-  Test model
-  Collaborators

Dynamic behavior of the server test



Example of server tests with Spring MockMvc

```
@WebMvcTest
@ContextConfiguration(classes = RestController.class)
class RestControllerTests {
    @Autowired
    private MockMvc mockMvc;
    @Autowired
    private ObjectMapper objectMapper;

    @Test
    void testGetAllObjects() throws Exception {
        List<Object> expectedResultList = new ArrayList<Object>(); // add the expected objects here
        ResultActions request = mockMvc.perform(get("/objects")).andDo(print()).andExpect(status().isOk());

        String response = request.andReturn().getResponse().getContentAsString();

        List<Object> actualResultList =
            Arrays.stream(objectMapper.readValue(response, Object[].class)).collect(Collectors.toList());

        assertEquals(expectedResultList, actualResultList, "Not all objects have been returned");
    }
}
```

Allows to mock the client

Performs a GET request on the actual server and checks that the response code is 200 (OK)

Response as (json) string

Map response to Java objects

Compare expected and actual response



L09E03 Testing a REST Server



Start exercise

Hard

Not started.

Due date in 7 days



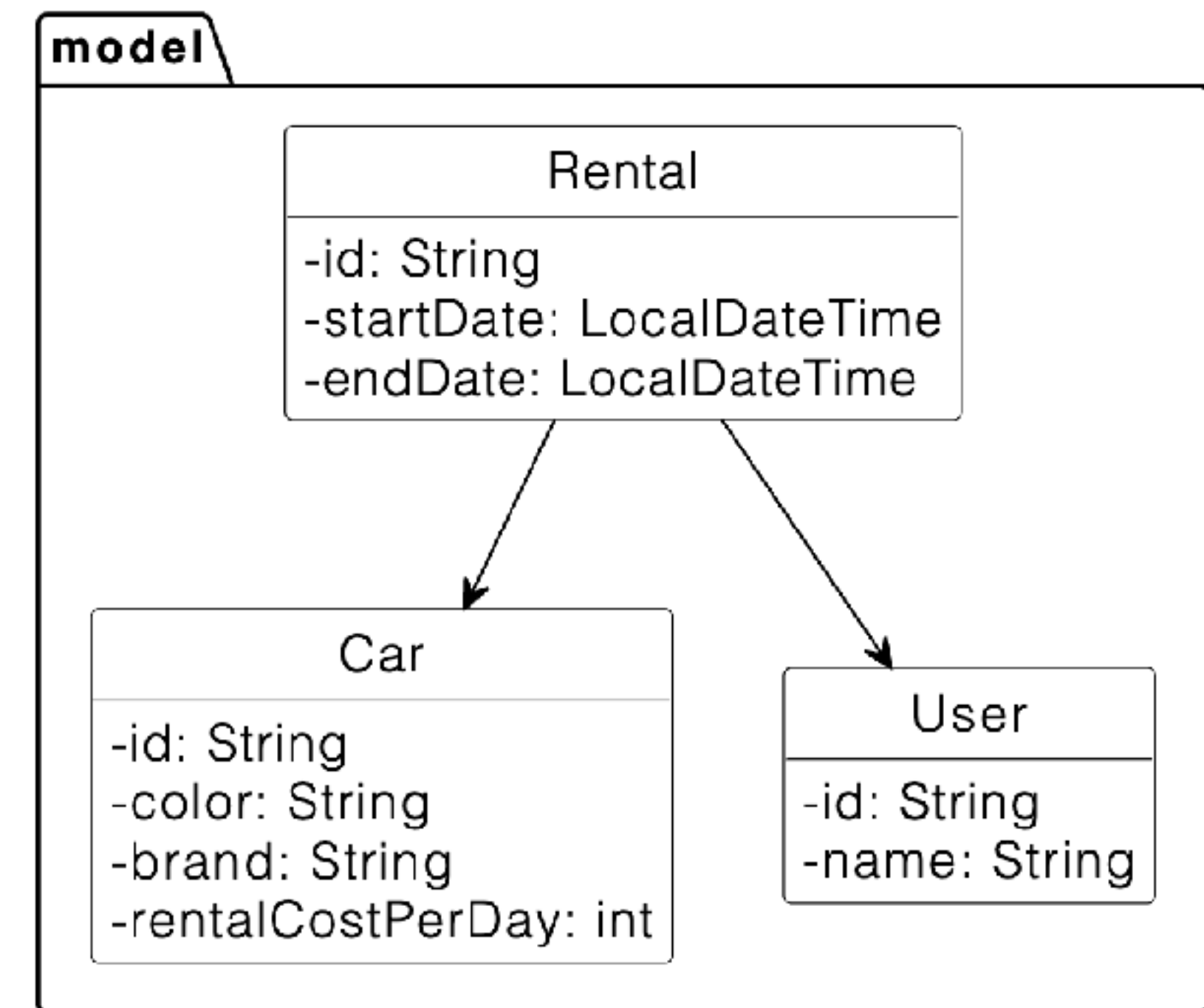
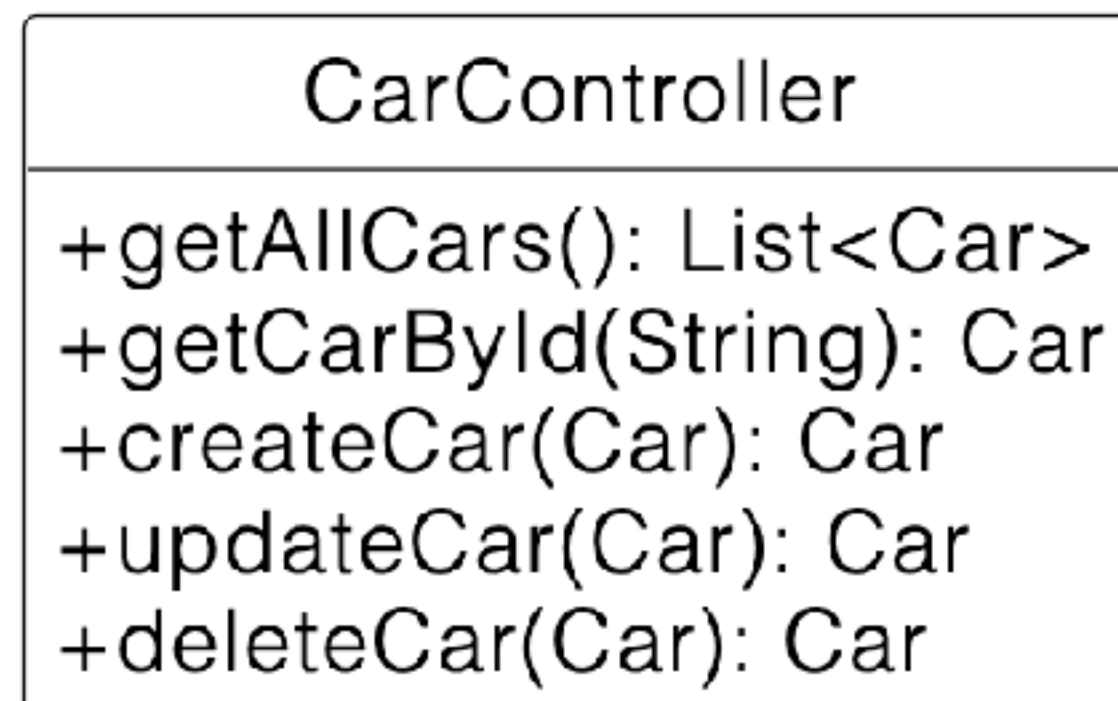
20 min



15 pts



- Problem statement
 - Test all requests in the **CarController** on the REST server
 - Mock the client using **MockMvc** in Spring



- Simple unit tests allow to test the state of an object or a subsystem
- The **mock object pattern** allows to unit test behavior
- We can achieve low coupling between the system and the tests with **dependency injection**
- Successful testing is difficult and there are many obstacles
- Follow the **best practices** and **modern test principles**
- There are many examples how to write integration tests for complex setups of distributed systems
- Are there other patterns applicable to testing?
 - Meszaros describes 68 testing patterns: Gerard Meszaros: xUnit Test Patterns – Refactoring Test Code. Martin Fowler Signature Series, Addison-Wesley, 2007

- Matthew Brown & Eli Tapolcsanyi: Mock Object Patterns. In Proceedings of the 10th Conference on Pattern Languages of Programs, 2003. Published online: <http://hillside.net/plop/plop2003/papers.html>
- Martin Fowler: Inversion of Control Containers and the Dependency Injection pattern. <http://martinfowler.com/articles/injection.html>
- Google Guice Framework: <https://github.com/google/guice>
- Gerard Meszaros: xUnit Test Patterns – Refactoring Test Code, Addison-Wesley, 2007
- Ralph Johnson and Brian Foote: Designing Reusable Classes. Journal of Object Oriented Programming, SIGS Publication Group, June-July 1988, pp.22-35.
- Martin Fowler, Dependency Injection: <http://martinfowler.com/articles/injection.html>
- <https://www.vogella.com/tutorials/DependencyInjection/article.html>
- <https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring>
- <https://www.vogella.com/tutorials/SpringDependencyInjection/article.html>
- <https://phauer.com/2019/modern-best-practices-testing-java>