Patterns in Software Engineering

08 Testing Patterns I

Stephan Krusche

09 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	Microservice Patterns I
11	30.01.23	Microservice Patterns II
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 have first experiences with the mock object pattern

L00E10 Mock Object Pattern

- **Learning goals: at the end of this lecture you are able to**
  - Apply the mock object pattern using EasyMock and Mockito
  - Explain test driven development
  - Test code based on common test patterns
  - Apply test patterns to concrete situations
  - Differentiate between the testing patterns

Outline

➡ Testing

- Mock object pattern
  - EasyMock
  - Mockito
- Test driven development
- Reflection test pattern
- Four stage testing pattern
- Testing patterns for MVC

What constitutes successful testing?

- The purpose of testing is the generation of failures
- Two ways to express the success of testing a component

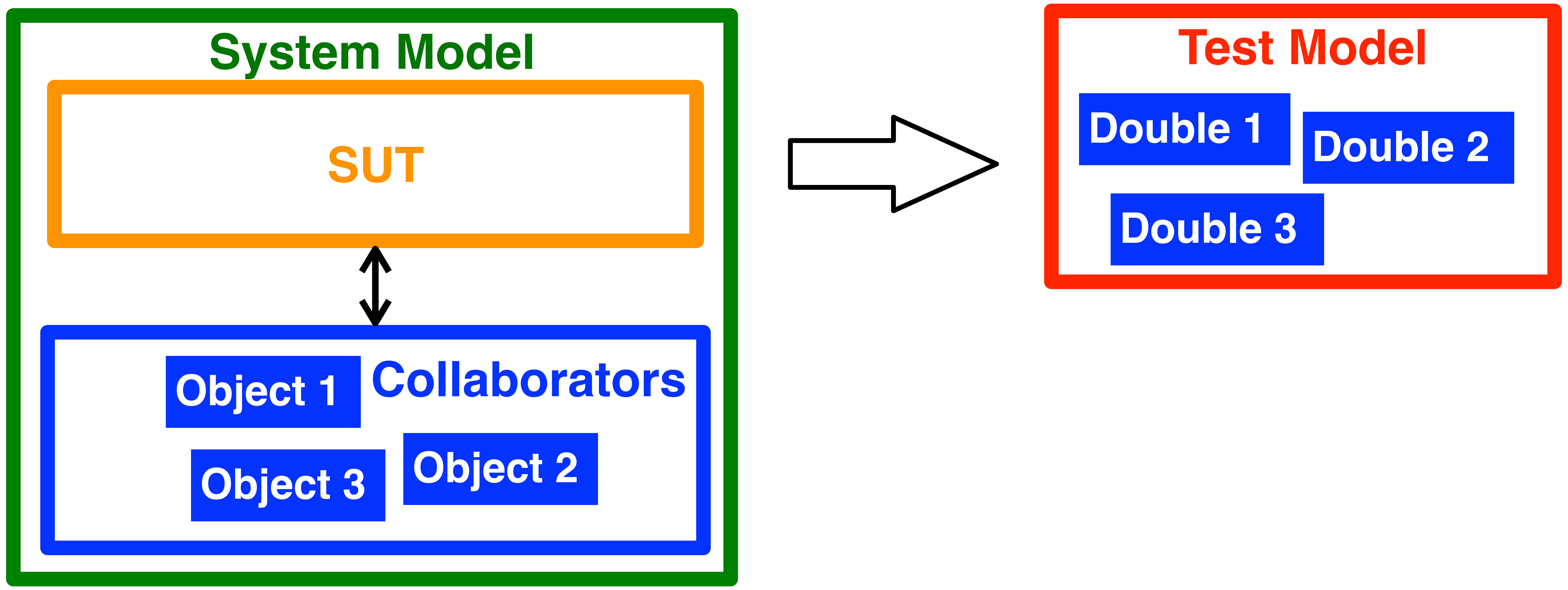
1. The test was **successful** because it did **not** generate a failure
    - Commonly used by many programmers
    - The goal is to show the absence of failures
2. The test was **successful** because it generated a failure
    - Karl Popper: the goal is the falsification of a model
    - ”A theory in the empirical sciences can never be proven, but it can be falsified, meaning that it should be scrutinized by decisive experiments”

JUnit 5

- Further development of JUnit 4 (mostly backwards compatible)
- User guide: https://junit.org/junit5/docs/current/user-guide/
- New features and key updates (https://www.netcentric.biz/insights/2020/07/junit5-new-features.html)
  - Precise exception handling
  - Friendly display names
  - Group assertions
  - Conditional, repeated, parameterized and dynamic tests
  - Meta annotations
  - Parallel tests, un-public, lambda expressions, 3rd party integrations, nests test classes, …

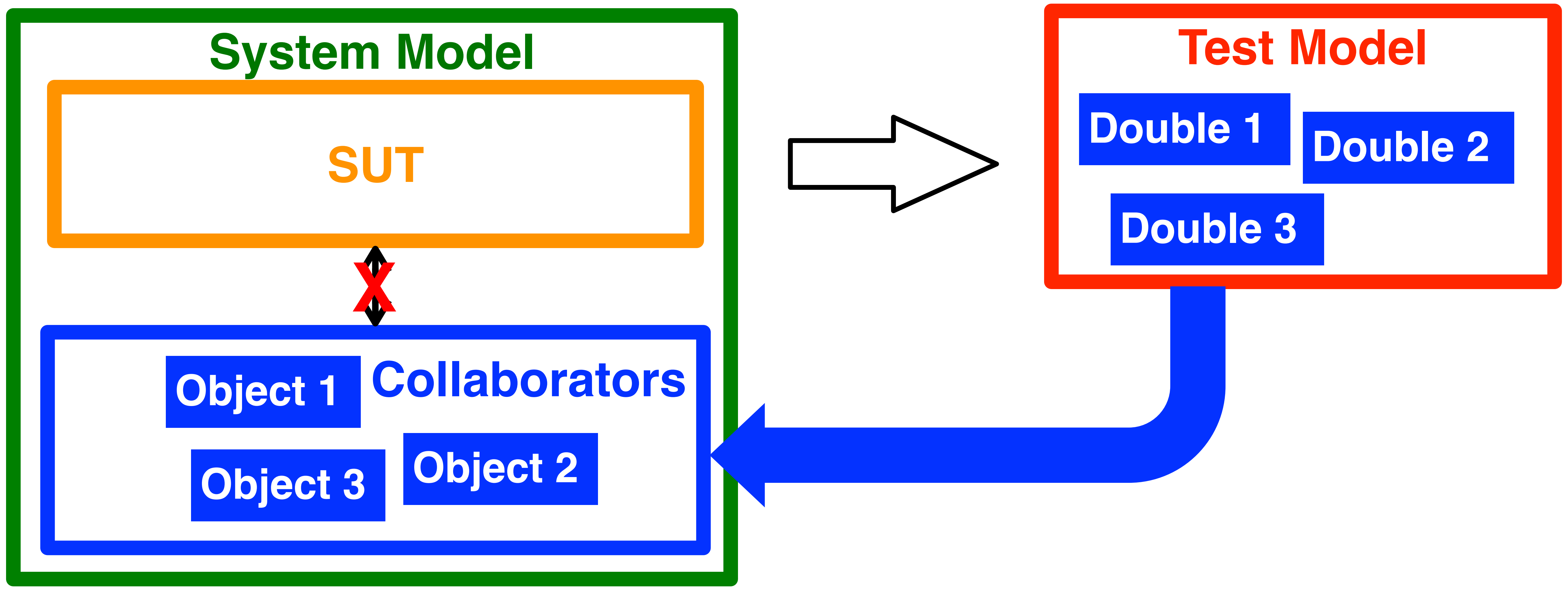
Object oriented test modeling (review)

- Start with the **system model**
- The system contains the **SUT** (system under test)
- The **SUT** does not exist in isolation, it interacts with other participating objects in the system model that are not yet implemented: **collaborators**
- The **test model** is derived from the **SUT**
- To be able to interact with **collaborators**, we add objects to the **test model**
- These are called **test doubles**

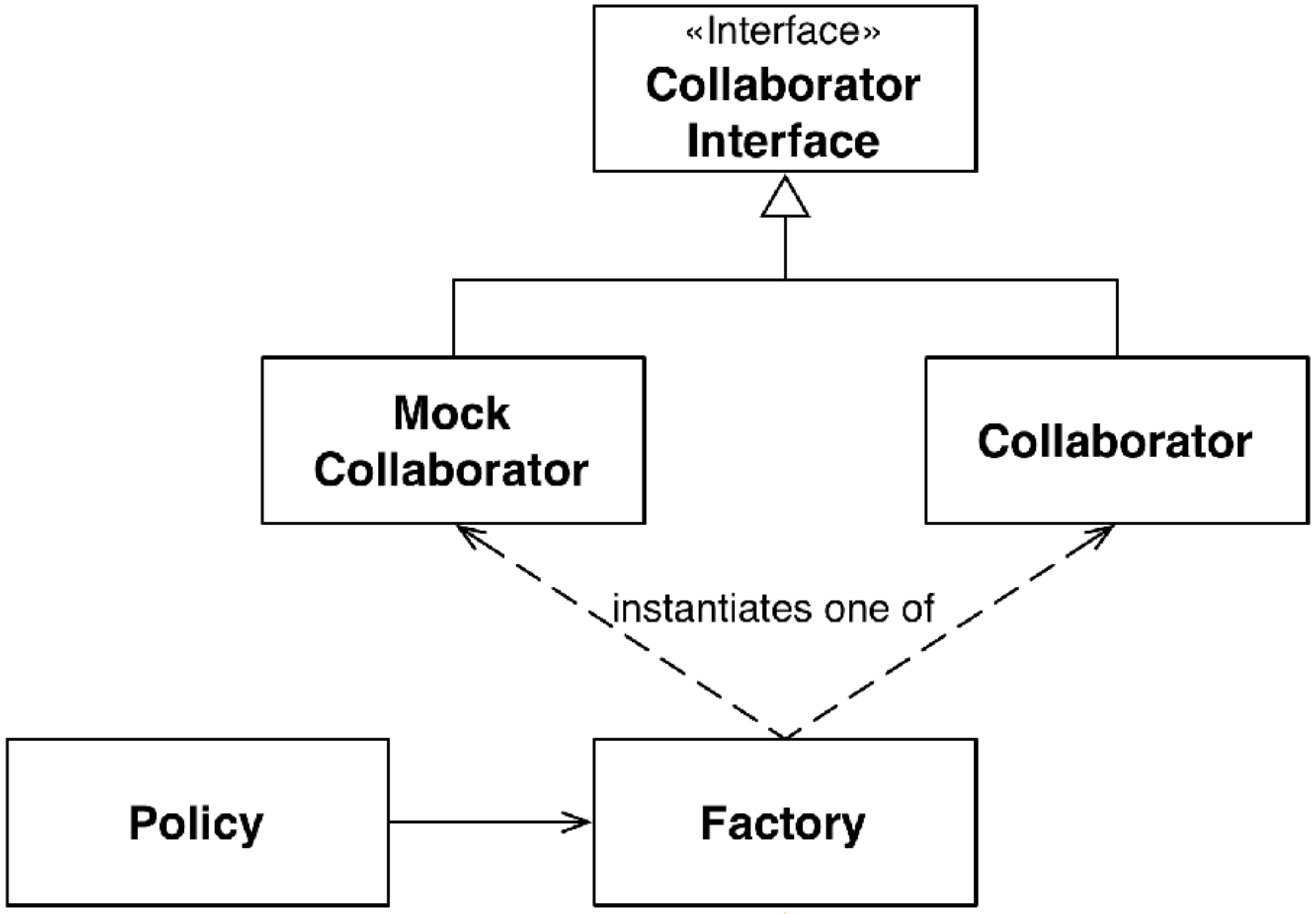


Object oriented test modeling (review)

- Start with the **system model**
- The system contains the **SUT** (system under test)
- The **SUT** does not exist in isolation, it interacts with other participating objects in the system model that are not yet implemented: **collaborators**
- The **test model** is derived from the **SUT**
- To be able to interact with **collaborators**, we add objects to the **test model**
- These are called **test doubles** (substitutes for the **collaborators** during testing)



Mock object pattern (review)

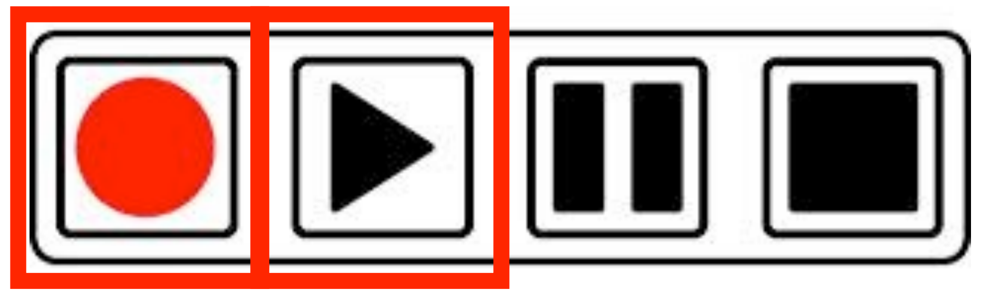


Framework, e.g. EasyMock / Mockito

- A **mock object** replaces the behavior of a real object called the collaborator and returns hard-coded values
- A mock object can be created at startup time with the factory pattern (not covered in the lecture, look it up in Gamma's book)
- Mock objects can be used for testing the state of individual objects as well as the interaction between objects
- The use of mock objects is based on the **record play metaphor**

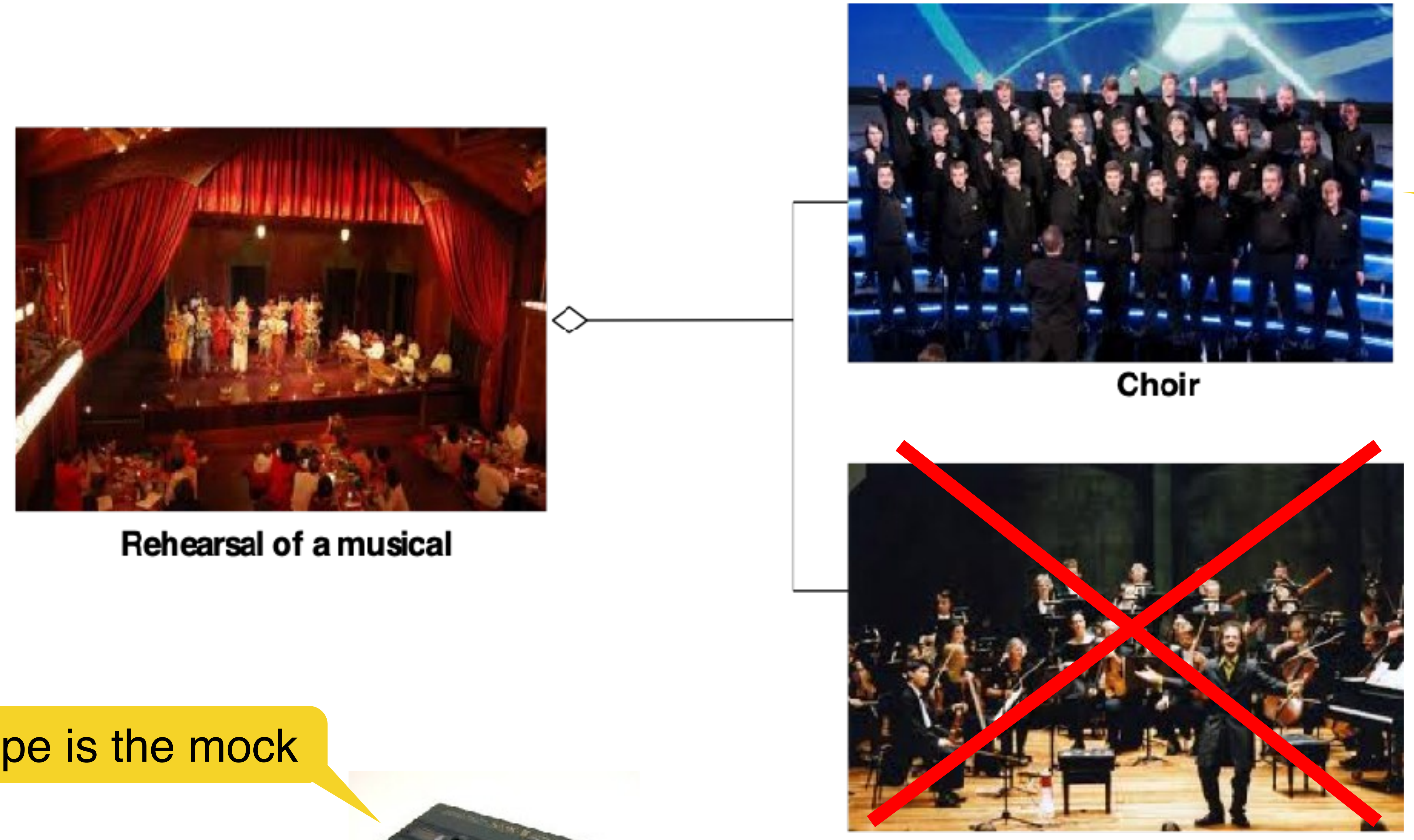
Step 2: Play

Step 1: Record



Record play metaphor (review)

Assume you want to perform a musical, which requires an orchestra and a choir. Most of the time the orchestra will not be available (too expensive), when the choir practices. But the choir needs to be accompanied by the music played by the orchestra when rehearsing the musical:



During rehearsal the choir must sing in sync with the music on the mock (tape)

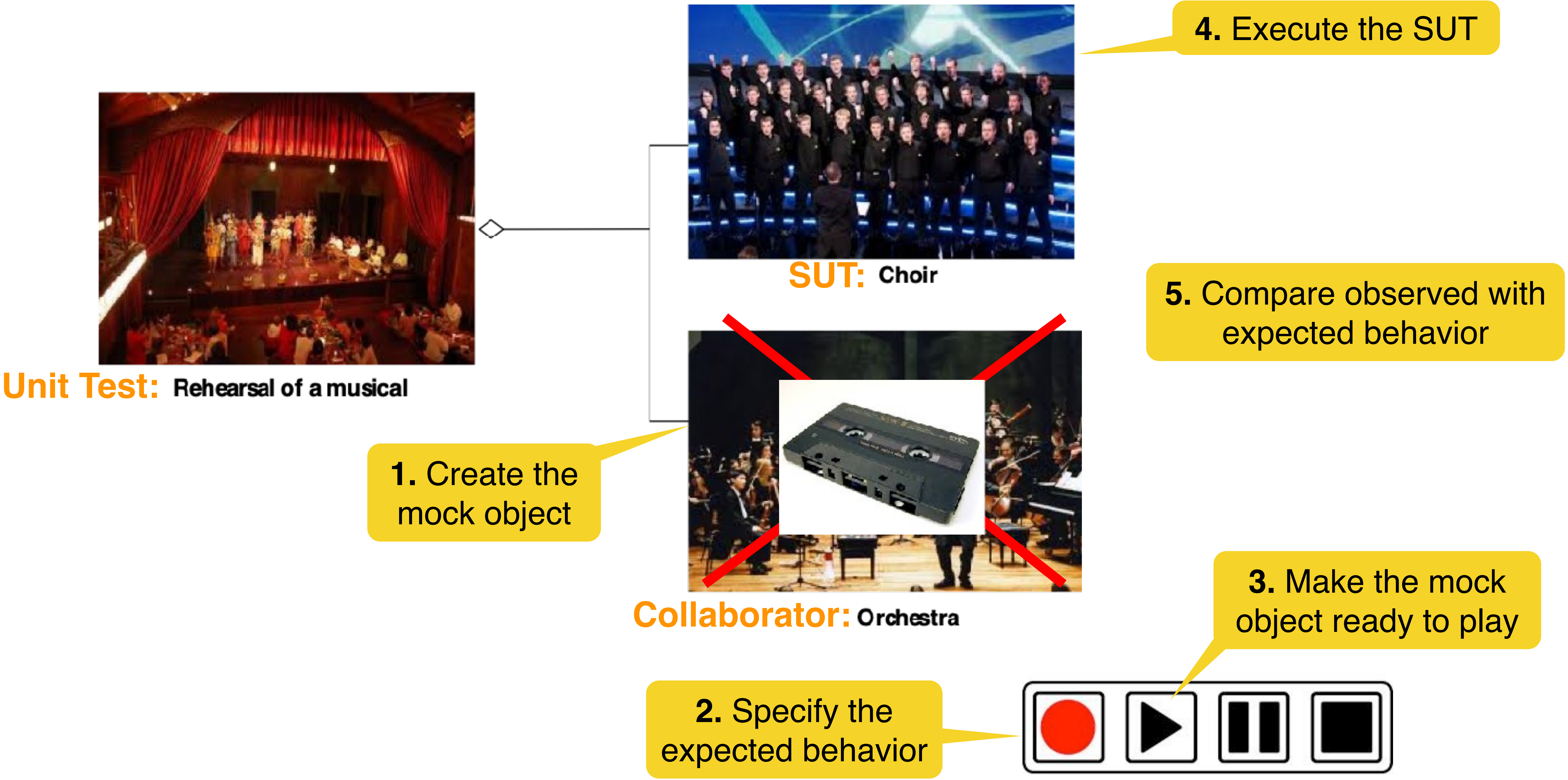
During rehearsal the music on the mock is played

The tape is the mock

The orchestra records the music onto the mock

Record play metaphor for mock objects (review)

Mock objects are proxy collaborators in tests where the real collaborators are not available



Unit Test: Rehearsal of a musical

SUT: Choir

Collaborator: Orchestra

1. Create the mock object

2. Specify the expected behavior

3. Make the mock object ready to play

4. Execute the SUT

5. Compare observed with expected behavior

EasyMock

- Open source testing framework for Java
- Uses annotations for test subjects (=**SUT**) and mocks

```
@TestSubject
private ClassUnderTest classUnderTest = new ClassUnderTest();

@Mock
private Collaborator mock;
```

- Specification of the behavior

```
expect(mock.invoke(parameter)).andReturn(42);
```

- Make the mock ready to play

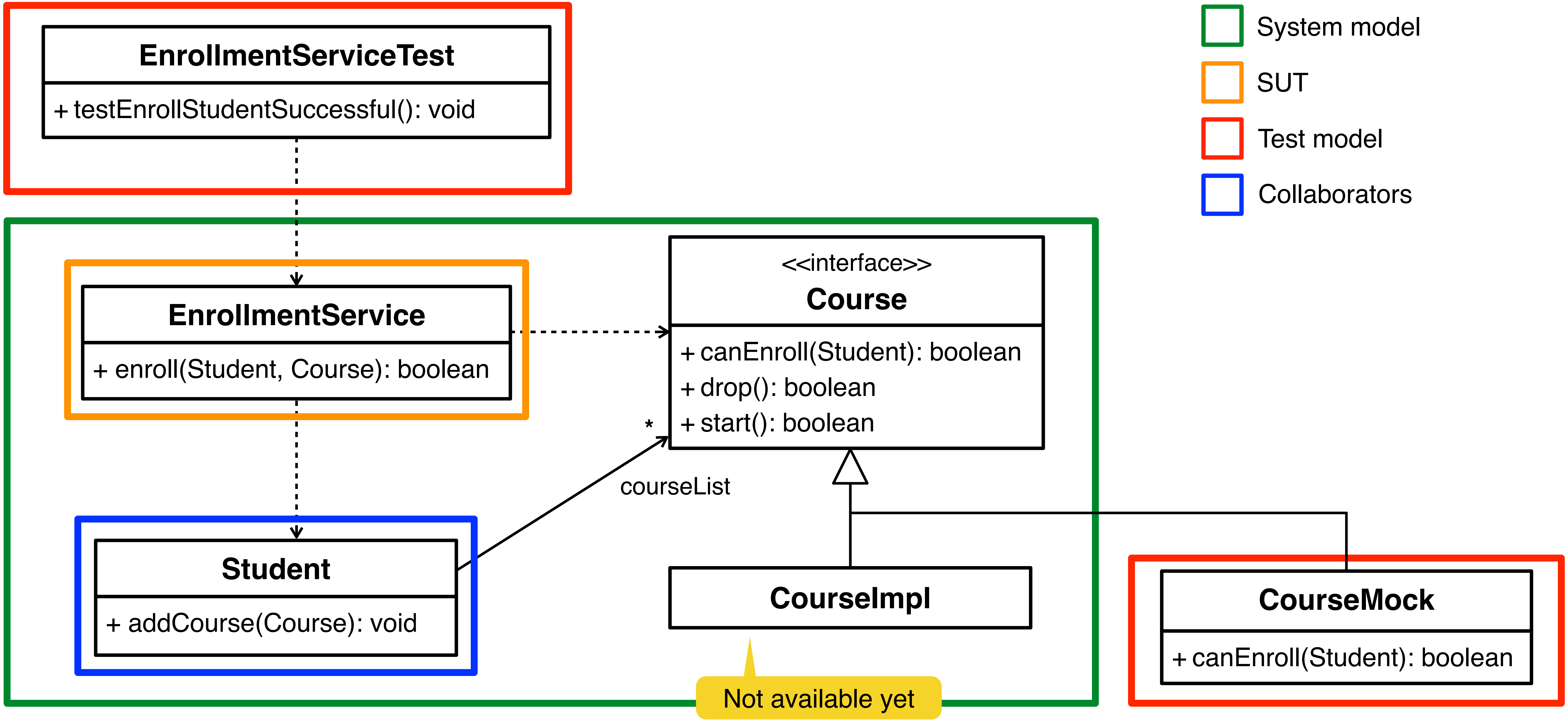
```
replay(mock);
```

- Make sure the mock has actually been called in the test (additional assertion)

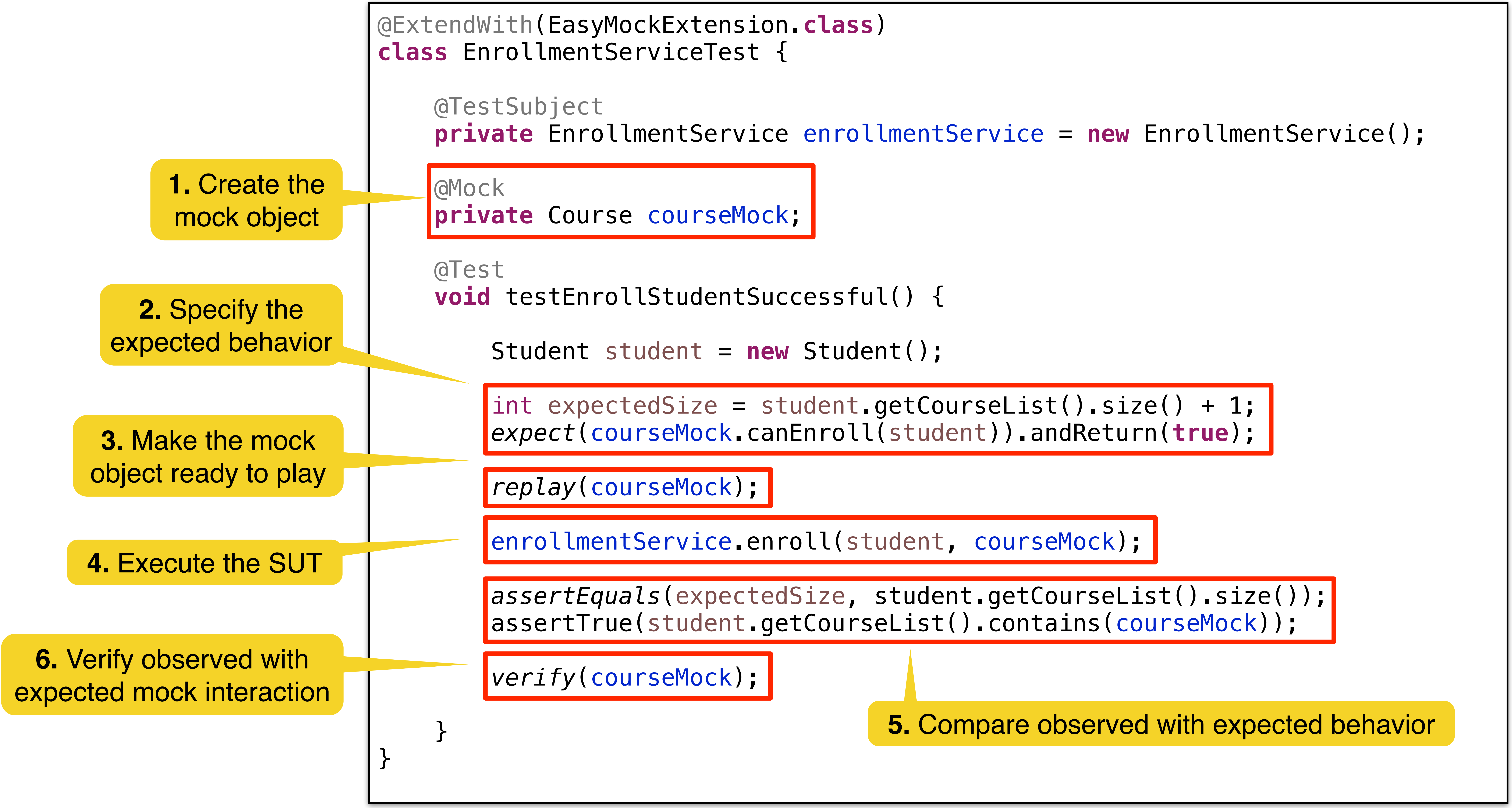
```
verify(mock);
```

- Documentation: http://easymock.org/user-guide.html

Example: University app with a mock object



Unit test for enrolling students with EasyMock



```java
@ExtendWith(EasyMockExtension.class)
class EnrollmentServiceTest {

    @TestSubject
    private EnrollmentService enrollmentService = new EnrollmentService();

    @Mock
    private Course courseMock;

    @Test
    void testEnrollStudentSuccessful() {

        Student student = new Student();

        int expectedSize = student.getCourseList().size() + 1;
        expect(courseMock.canEnroll(student)).andReturn(true);

        replay(courseMock);

        enrollmentService.enroll(student, courseMock);

        assertEquals(expectedSize, student.getCourseList().size());
        assertTrue(student.getCourseList().contains(courseMock));

        verify(courseMock);

    }
}
```

1. Create the mock object

2. Specify the expected behavior

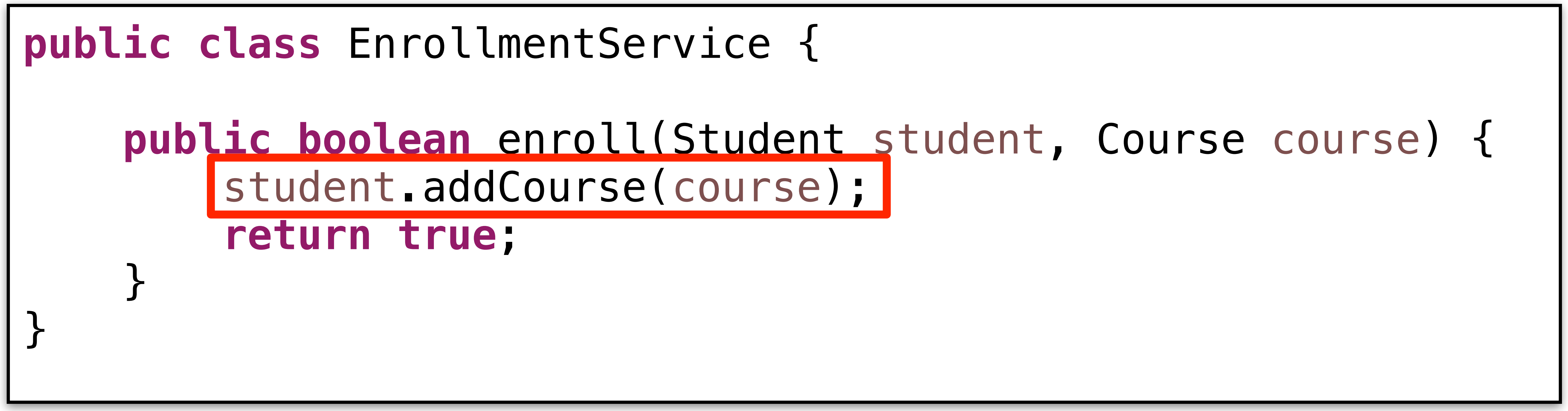
3. Make the mock object ready to play

4. Execute the SUT

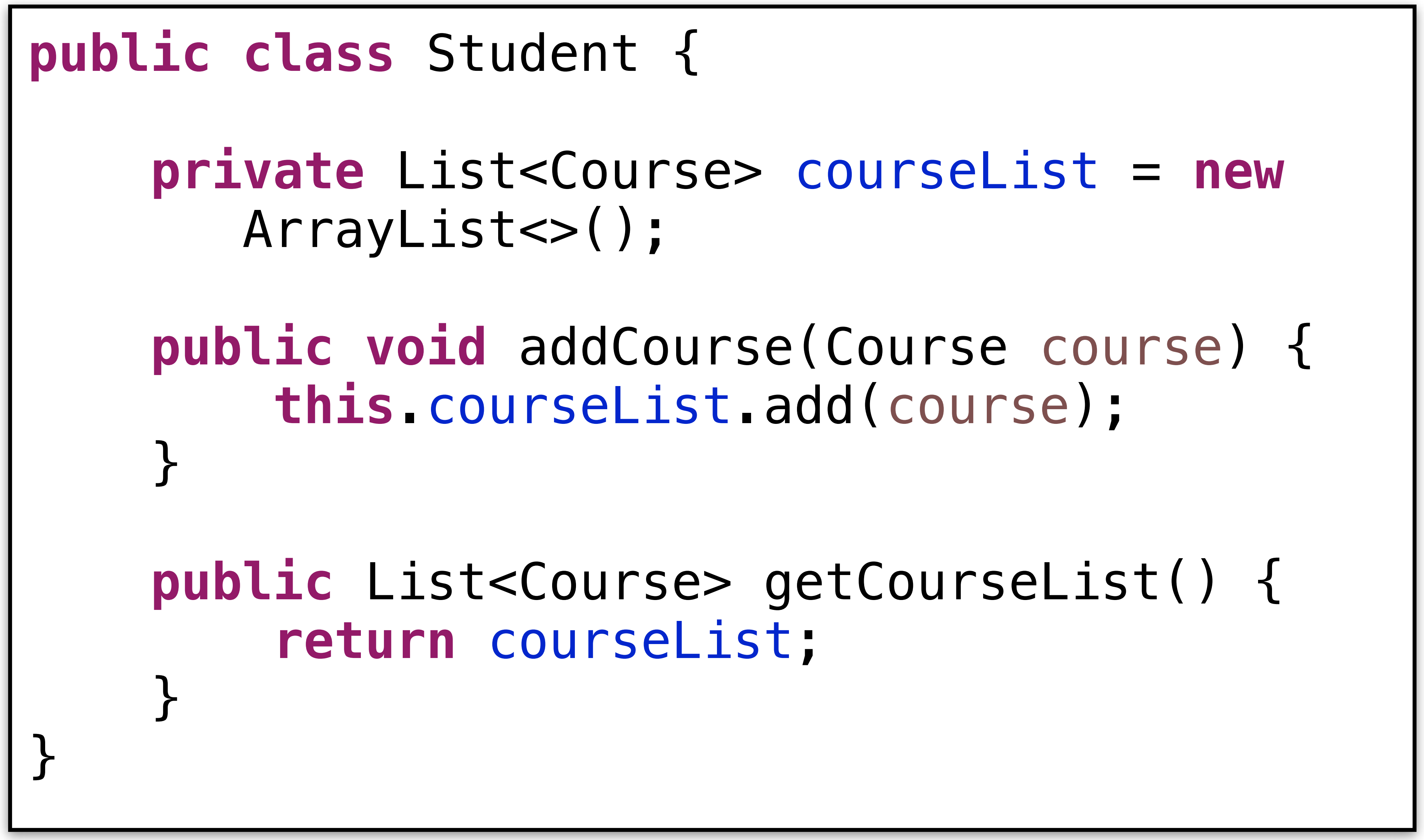
5. Compare observed with expected behavior

6. Verify observed with expected mock interaction

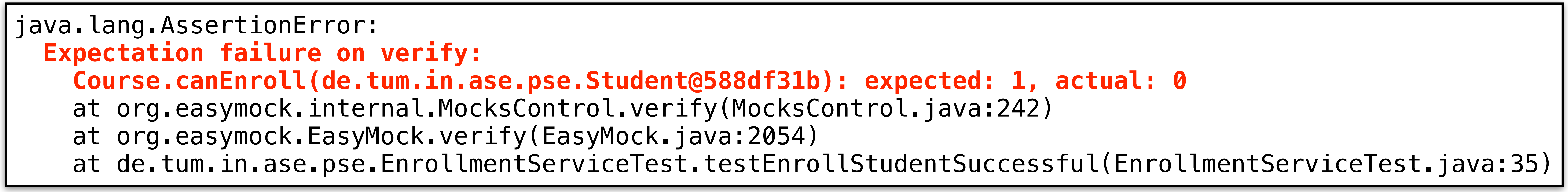
Example: wrong SUT code leads to a **failing** test



```java
public class EnrollmentService {

    public boolean enroll(Student student, Course course) {
        student.addCourse(course);
        return true;
    }
}
```



```java
public class Student {

    private List<Course> courseList = new
       ArrayList<>();

    public void addCourse(Course course) {
        this.courseList.add(course);
    }

    public List<Course> getCourseList() {
        return courseList;
    }
}
```



```
java.lang.AssertionError:
  Expectation failure on verify:
    Course.canEnroll(de.tum.in.ase.pse.Student@588df31b): expected: 1, actual: 0
    at org.easymock.internal.MocksControl.verify(MocksControl.java:242)
    at org.easymock.EasyMock.verify(EasyMock.java:2054)
    at de.tum.in.ase.pse.EnrollmentServiceTest.testEnrollStudentSuccessful(EnrollmentServiceTest.java:35)
```

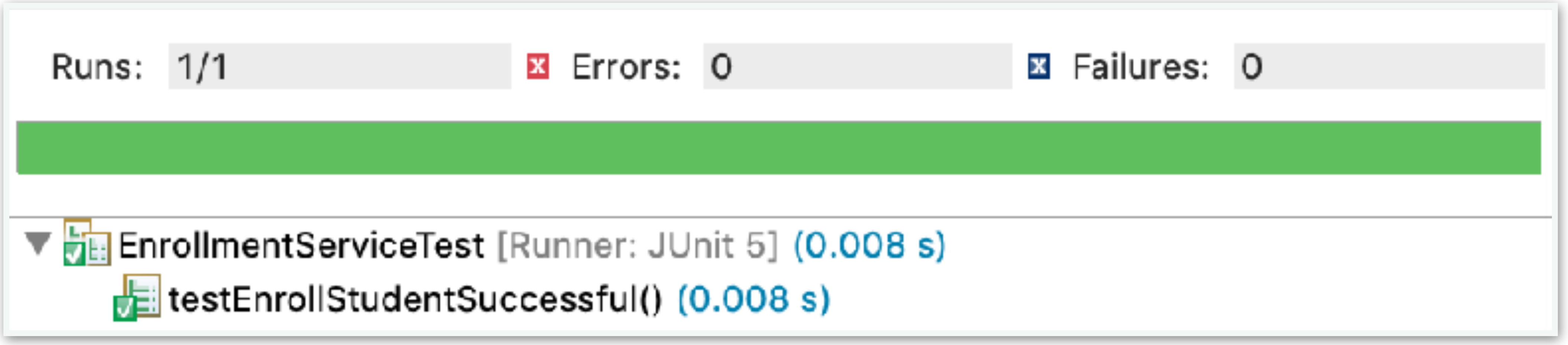
Problem: even if all **`assertions`** pass, the test fails

Reason: the method **`canEnroll`** on **`Course`** was not invoked

Example: correct SUT code leads to a **passing** test

```java
public class EnrollmentService {

    public boolean enroll(Student student, Course course) {
        if (course.canEnroll(student)) {
            student.addCourse(course);
            return true;
        }
        return false;
    }
}
```

```java
public class Student {

    private List<Course> courseList = new
       ArrayList<>();

    public void addCourse(Course course) {
        this.courseList.add(course);
    }

    public List<Course> getCourseList() {
        return courseList;
    }
}
```



```
Runs: 1/1     Errors: 0     Failures: 0

▼ EnrollmentServiceTest [Runner: JUnit 5] (0.008 s)
     testEnrollStudentSuccessful() (0.008 s)
```

Even if the implementation of **`Course`** is missing, we can test if the implementation of **`EnrollmentService`** is correct

Nice vs. default vs. strict mocks

- EasyMock offers 3 different mock version

1. **`@Mock(MockType.NICE)`**
    - Allows all method calls and returns appropriate empty values (**`0`**, **`null`**, or **`false`**)
2. **`@Mock`** Default mock
    - Throws an **`AssertionError`** for unexpected method calls
    - Does not check the order of method calls
3. **`@Mock(MockType.STRICT)`**
    - Checks the order of method calls

Example: **default mock** throws an error

```java
@ExtendWith(EasyMockExtension.class)
class EnrollmentServiceTest {

    @TestSubject
    private EnrollmentService enrollmentService = new EnrollmentService();

    @Mock
    private Course courseMock;

    @Test
    void testEnrollStudentSuccessful() { ... }
}
```

```java
public class Student {

    private List<Course> courseList = new
       ArrayList<>();

    public void addCourse(Course course) {
        this.courseList.add(course);
        course.start();
    }

    public List<Course> getCourseList() {
        return courseList;
    }
}
```

Unexpected method call

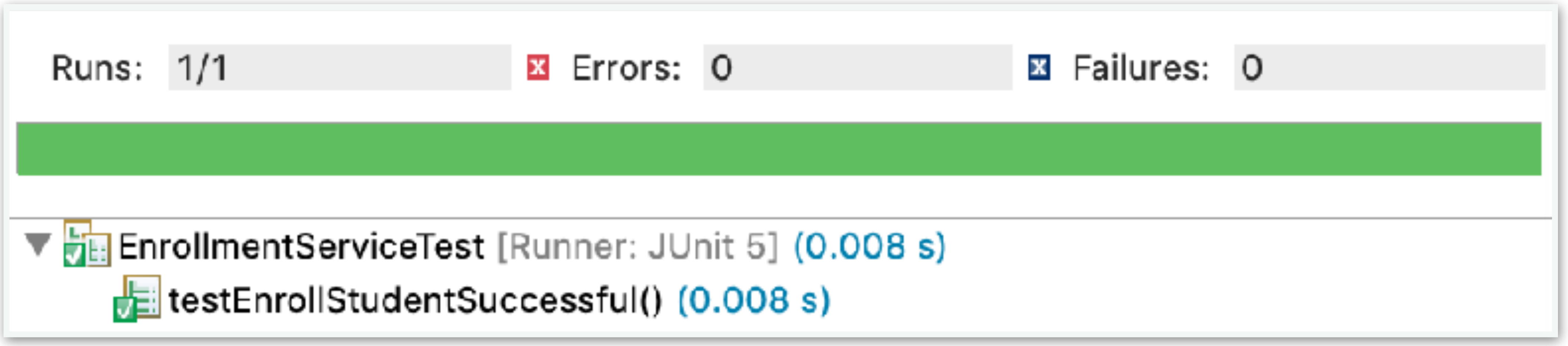
```
java.lang.AssertionError:
  Unexpected method call Course.start():
    at org.easymock.internal.MockInvocationHandler.invoke(MockInvocationHandler.java:44)
    at org.easymock.internal.ObjectMethodsFilter.invoke(ObjectMethodsFilter.java:101)
    at com.sun.proxy.$Proxy11.start(Unknown Source)
    at de.tum.in.ase.pse.Student.addCourse(Student.java:12)
    at de.tum.in.ase.pse.EnrollmentService.enroll(EnrollmentService.java:8)
    at de.tum.in.ase.pse.EnrollmentServiceTest.testEnrollStudentSuccessful(EnrollmentServiceTest.java:30)
```

Example: **nice mock** does not throw

```
@ExtendWith(EasyMockExtension.class)
class EnrollmentServiceTest {

    @TestSubject
    private EnrollmentService enrollmentService = new EnrollmentService();

    @Mock(type = MockType.NICE)
    private Course courseMock;

    @Test
    void testEnrollStudentSuccessful() { ... }
}
```

```
public class Student {

    private List<Course> courseList = new
        ArrayList<>();

    public void addCourse(Course course) {
        this.courseList.add(course);
        course.start();
    }

    public List<Course> getCourseList() {
        return courseList;
    }
}
```

Unexpected method call



```
Runs:  1/1        Errors:  0        Failures:  0

▼ EnrollmentServiceTest [Runner: JUnit 5] (0.008 s)
    testEnrollStudentSuccessful() (0.008 s)
```

Example: **strict mock** throws an error

```java
@ExtendWith(EasyMockExtension.class)
class EnrollmentServiceTest {

    @TestSubject
    private EnrollmentService enrollmentService = new EnrollmentService();

    @Mock(type = MockType.STRICT)
    private Course courseMock;

    @Test
    void testEnrollStudentSuccessful() {

        Student student = new Student();
        int expectedSize = student.getCourseList().size() + 1;

        expect(courseMock.start()).andReturn(true);
        expect(courseMock.canEnroll(student)).andReturn(true);

        replay(courseMock);

        enrollmentService.enroll(student, courseMock);

        assertEquals(expectedSize, student.getCourseList().size());
        assertTrue(student.getCourseList().contains(courseMock));

        verify(courseMock);
    }
}
```

Strict mock

Wrong order

```java
public class Student {

    private List<Course> courseList = new
       ArrayList<>();

    public void addCourse(Course course) {
        this.courseList.add(course);
        course.start();
    }

    public List<Course> getCourseList() {
        return courseList;
    }
}
```

```
java.lang.AssertionError:
  Unexpected method call Course.canEnroll(de.tum.in.ase.pse.Student@f4168b8):
    Course.start(): expected: 1, actual: 0
    at org.easymock.internal.MockInvocationHandler.invoke(MockInvocationHandler.java:44)
    at org.easymock.internal.ObjectMethodsFilter.invoke(ObjectMethodsFilter.java:101)
    at com.sun.proxy.$Proxy11.canEnroll(Unknown Source)
    at de.tum.in.ase.pse.EnrollmentService.enroll(EnrollmentService.java:7)
    at de.tum.in.ase.pse.EnrollmentServiceTest.testEnrollStudentSuccessful(EnrollmentServiceTest.java:32)
```

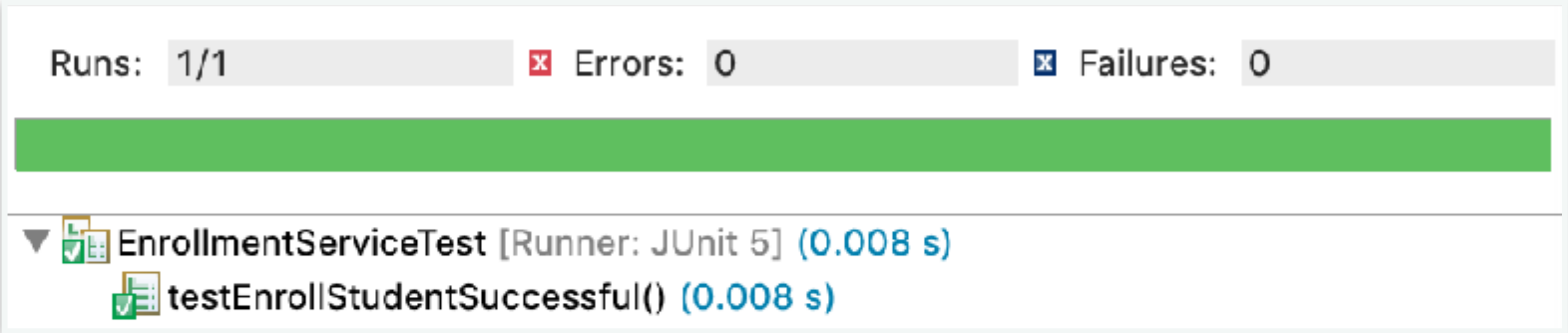
Example: **default mock** does **not** throw an error

```java
@ExtendWith(EasyMockExtension.class)
class EnrollmentServiceTest {

    @TestSubject
    private EnrollmentService enrollmentService = new EnrollmentService();

    @Mock
    private Course courseMock;

    @Test
    void testEnrollStudentSuccessful() {

        Student student = new Student();
        int expectedSize = student.getCourseList().size() + 1;

        expect(courseMock.start()).andReturn(true);
        expect(courseMock.canEnroll(student)).andReturn(true);

        replay(courseMock);

        enrollmentService.enroll(student, courseMock);

        assertEquals(expectedSize, student.getCourseList().size());
        assertTrue(student.getCourseList().contains(courseMock));

        verify(courseMock);
    }
}
```

Default mock

Wrong order

```java
public class Student {

    private List<Course> courseList = new
       ArrayList<>();

    public void addCourse(Course course) {
        this.courseList.add(course);
        course.start();
    }

    public List<Course> getCourseList() {
        return courseList;
    }
}
```



```
Runs: 1/1    Errors: 0    Failures: 0

EnrollmentServiceTest [Runner: JUnit 5] (0.008 s)
    testEnrollStudentSuccessful() (0.008 s)
```

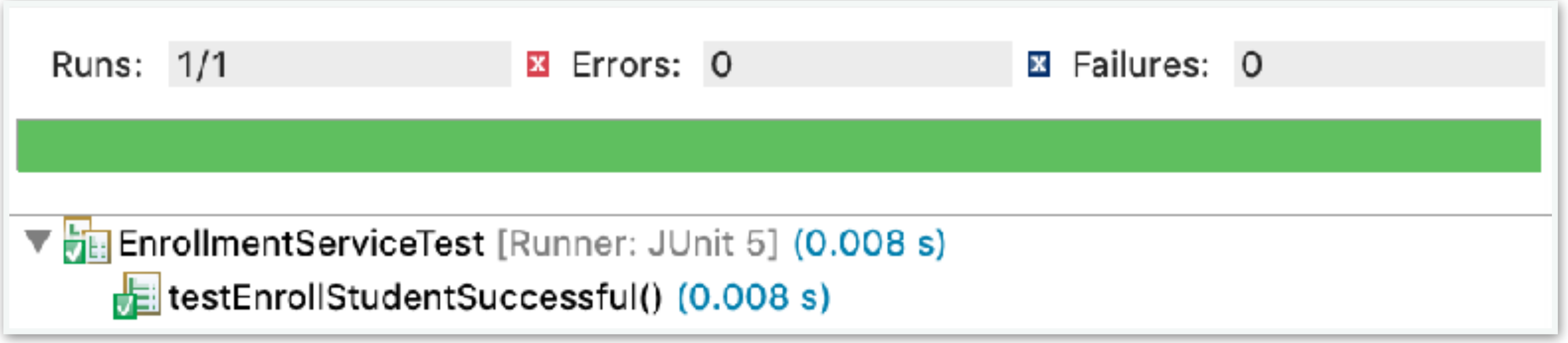
Example: **strict mock** does **not** throw an error

```java
@ExtendWith(EasyMockExtension.class)
class EnrollmentServiceTest {

    @TestSubject
    private EnrollmentService enrollmentService = new EnrollmentService();

    @Mock(type = MockType.STRICT)
    private Course courseMock;

    @Test
    void testEnrollStudentSuccessful() {

        Student student = new Student();
        int expectedSize = student.getCourseList().size() + 1;

        expect(courseMock.canEnroll(student)).andReturn(true);
        expect(courseMock.start()).andReturn(true);

        replay(courseMock);

        enrollmentService.enroll(student, courseMock);

        assertEquals(expectedSize, student.getCourseList().size());
        assertTrue(student.getCourseList().contains(courseMock));

        verify(courseMock);
    }
}
```

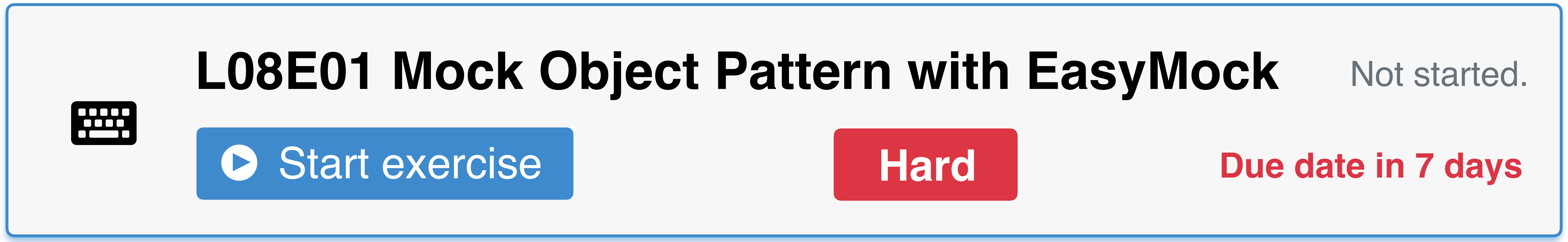
Strict mock

Correct order

```java
public class Student {

    private List<Course> courseList = new
       ArrayList<>();

    public void addCourse(Course course) {
        this.courseList.add(course);
        course.start();
    }

    public List<Course> getCourseList() {
        return courseList;
    }
}
```



```
Runs: 1/1    Errors: 0    Failures: 0

EnrollmentServiceTest [Runner: JUnit 5] (0.008 s)
    testEnrollStudentSuccessful() (0.008 s)
```



⌨ L08E01 Mock Object Pattern with EasyMock

Not started.

▶ Start exercise

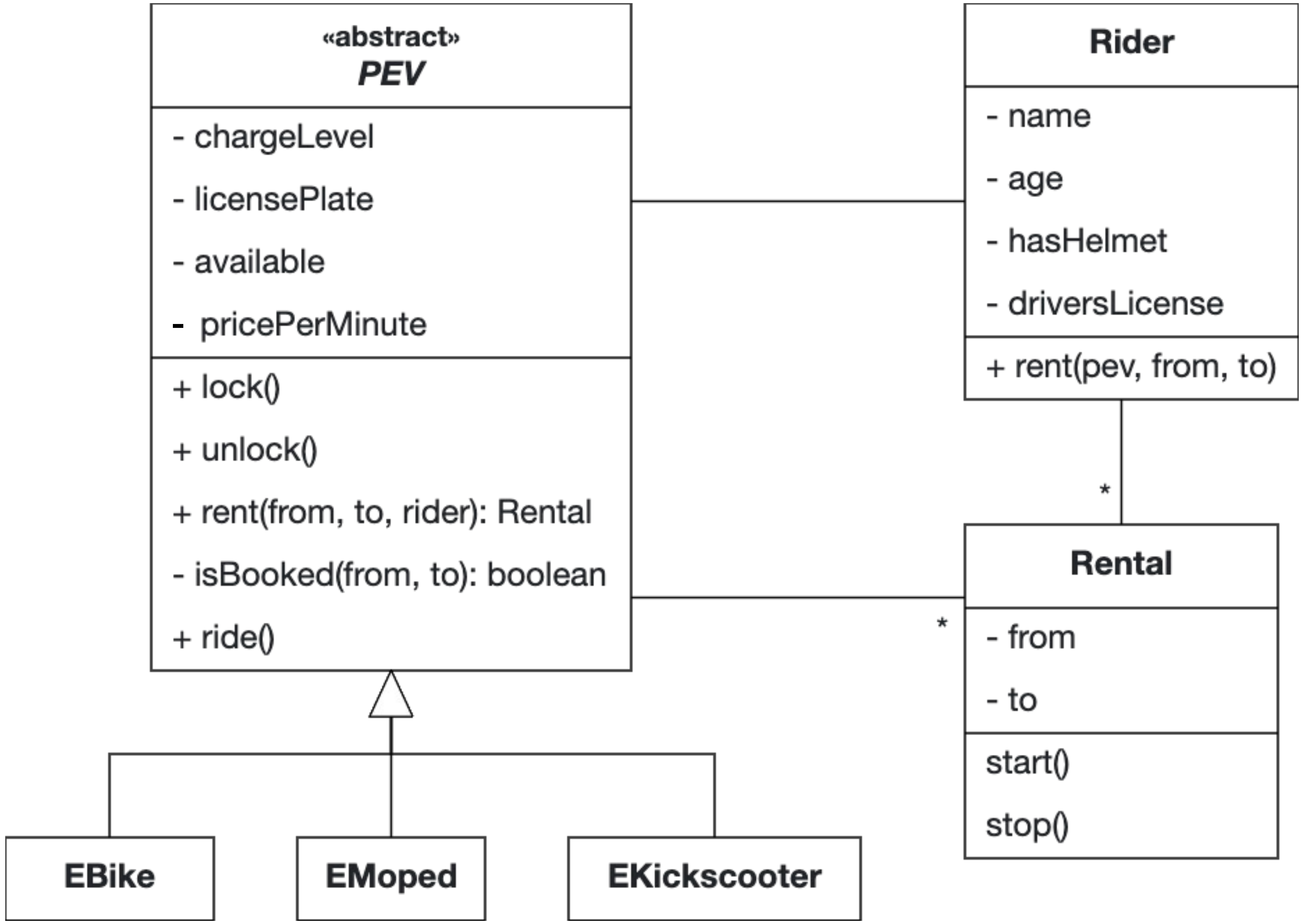
Hard

Due date in 7 days

🕓 20 min

🏆 14 pts

- Problem statement: PEV rentals **Navigation System**
- Advanced mocking techniques
  - Nice vs. default vs. strict mocks
  - Verify



«abstract»
PEV

- chargeLevel
- licensePlate
- available
- pricePerMinute

+ lock()
+ unlock()
+ rent(from, to, rider): Rental
- isBooked(from, to): boolean
+ ride()

Rider

- name
- age
- hasHelmet
- driversLicense

+ rent(pev, from, to)

Rental

- from
- to

start()
stop()

EBike

EMoped

EKickscooter

Hints for the presentation score for testing patterns

- Create a UML diagram that highlights the different parts in Apollon
  - System model
  - SUT
  - Test model including the mocks
  - Collaborators
- Use the following color scheme

Apollon supports coloring of classes

🟩 System model

🟧 SUT

🟥 Test model

🟦 Collaborators

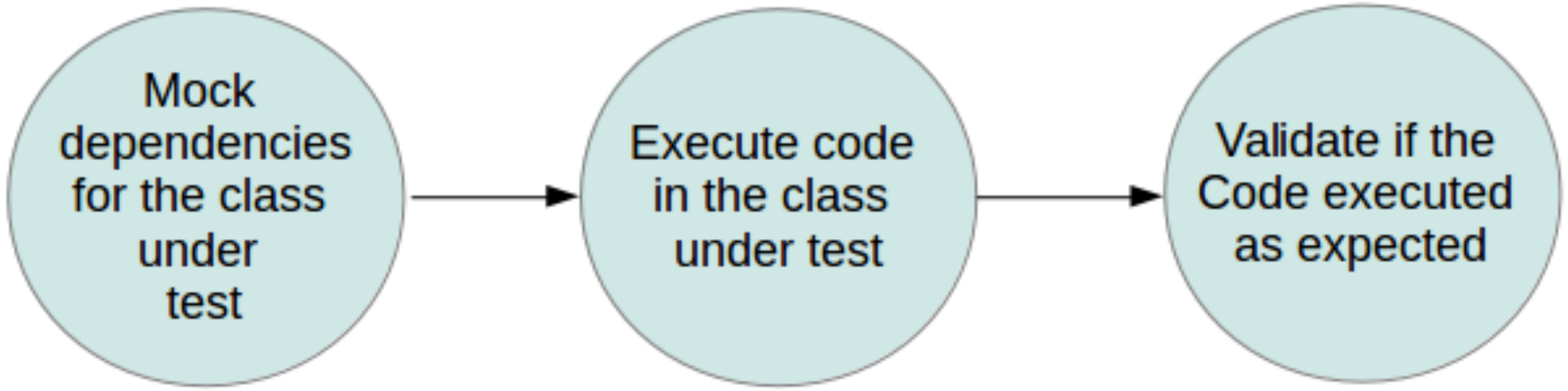
Outline

- Testing
- Mock object pattern
  - EasyMock
  - ➡ Mockito
- Test driven development
- Reflection test pattern
- Four stage testing pattern
- Testing patterns for MVC

Mockito

- Popular open source mocking framework for unit tests
- Allows the creation and configuration of mock objects for the purpose of
  - Test driven development (TDD)
  - Behavior driven development (BDD)
- Mockito simplifies the development of tests for classes with external dependencies
- Mockito began by expanding on the syntax and functionality of EasyMock
- https://site.mockito.org

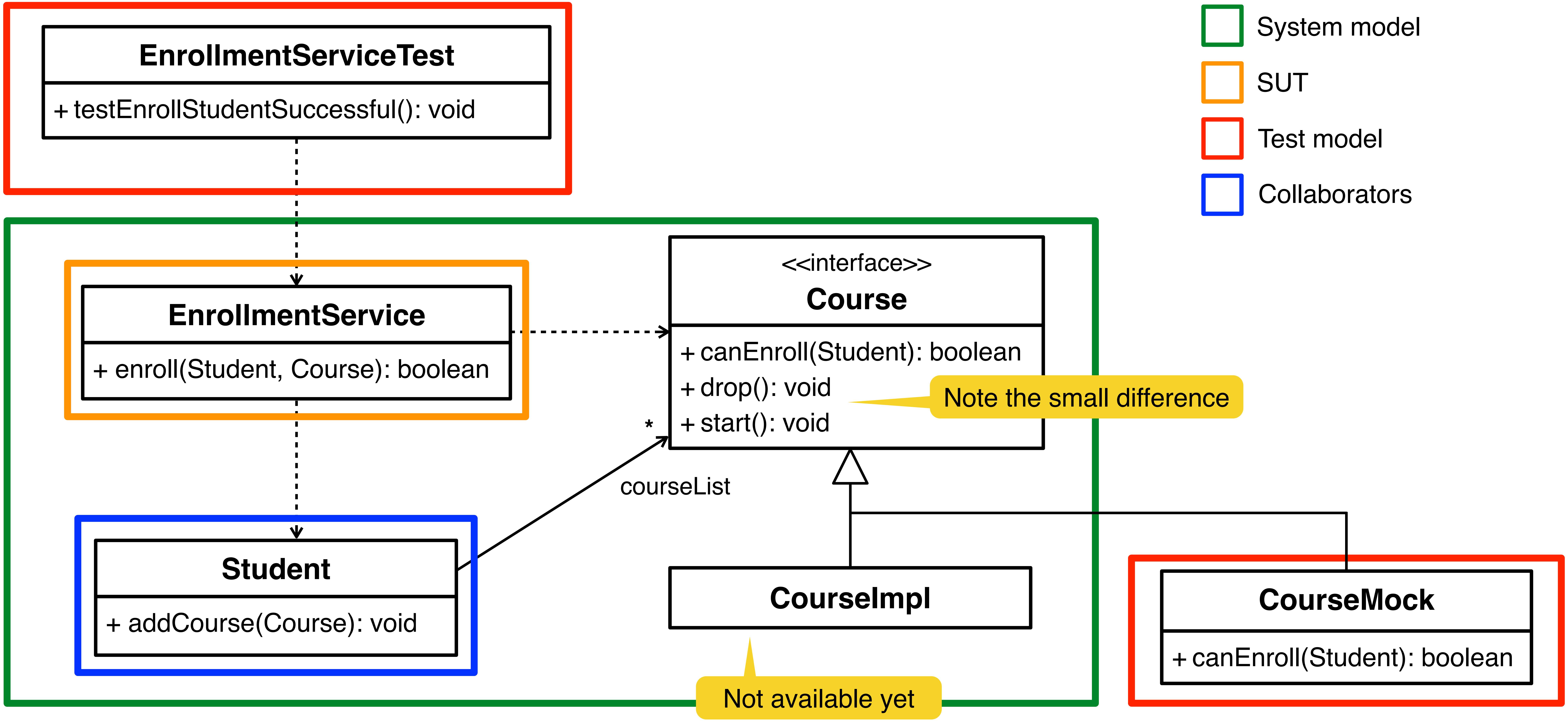
More about TDD later


Mock dependencies for the class under test
Execute code in the class under test
Validate if the Code executed as expected

Main features

- **`mock()/@Mock:`** full mocking
- **`spy()/@Spy:`** partial mocking
- **`@InjectMocks:`** automatically inject fields annotated with **`@Spy`** or **`@Mock`**
- **`verify():`** check that methods were called with given arguments
  - Can use flexible argument matching, for example any expression via the **`any()`**
  - Capture what arguments were called using **`@Captor`**

Example: University app with a mock object


EnrollmentServiceTest
+ testEnrollStudentSuccessful(): void
EnrollmentService
+ enroll(Student, Course): boolean
<<interface>>
Course
+ canEnroll(Student): boolean
+ drop(): void
+ start(): void
*
courseList
Student
+ addCourse(Course): void
CourseImpl
CourseMock
+ canEnroll(Student): boolean

Note the small difference

Not available yet

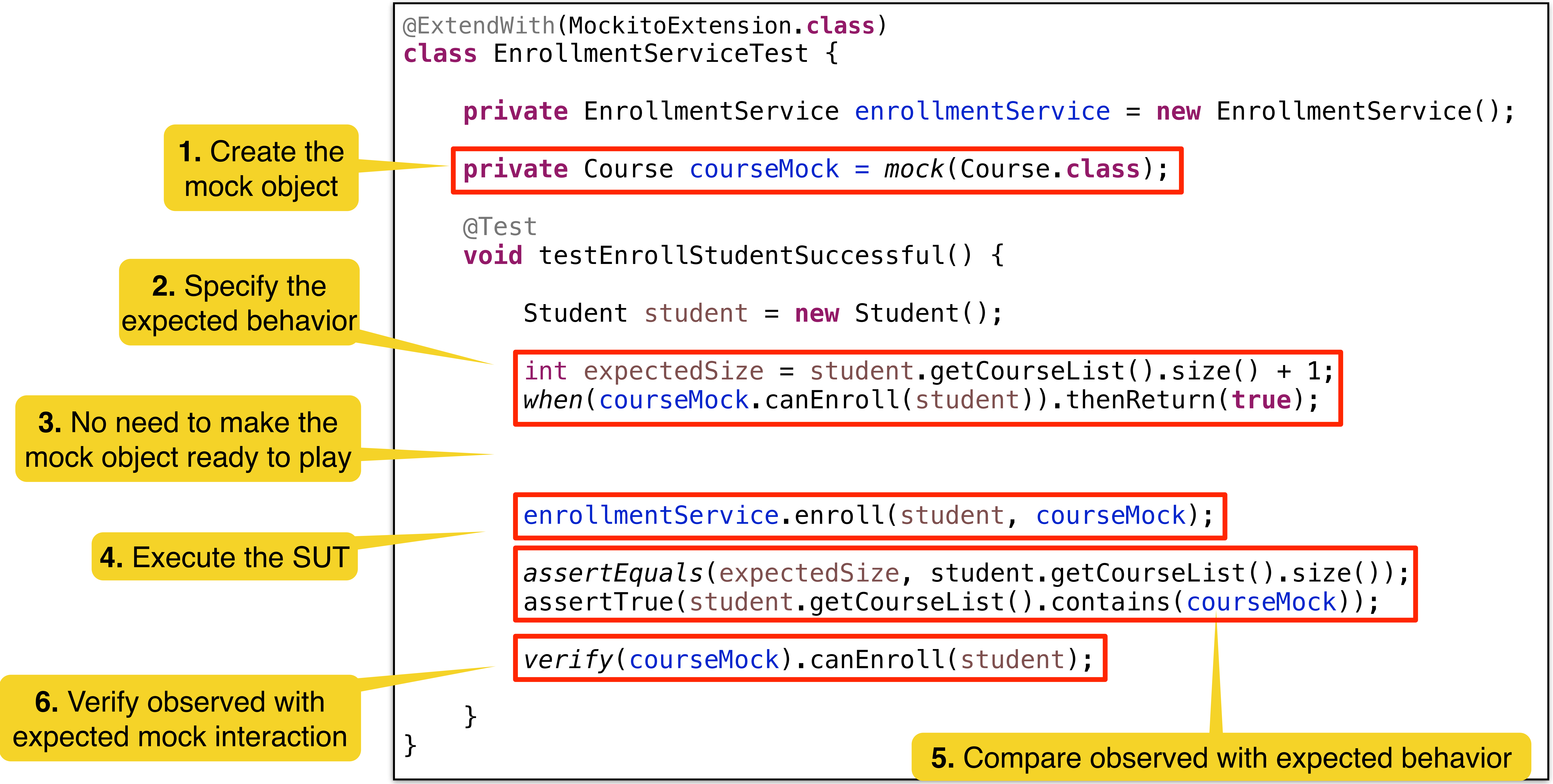
🟩 System model

🟧 SUT

🟥 Test model

🟦 Collaborators

Unit test for enrolling students with Mockito



```
@ExtendWith(MockitoExtension.class)
class EnrollmentServiceTest {

    private EnrollmentService enrollmentService = new EnrollmentService();

    private Course courseMock = mock(Course.class);

    @Test
    void testEnrollStudentSuccessful() {

        Student student = new Student();

        int expectedSize = student.getCourseList().size() + 1;
        when(courseMock.canEnroll(student)).thenReturn(true);


        enrollmentService.enroll(student, courseMock);

        assertEquals(expectedSize, student.getCourseList().size());
        assertTrue(student.getCourseList().contains(courseMock));

        verify(courseMock).canEnroll(student);

    }
}
```

1. Create the mock object

2. Specify the expected behavior

3. No need to make the mock object ready to play

4. Execute the SUT

5. Compare observed with expected behavior

6. Verify observed with expected mock interaction

Example: wrong SUT code leads to a failing test

```java
public class EnrollmentService {

    public boolean enroll(Student student, Course course) {
        student.addCourse(course);
        return true;
    }
}
```

```java
public class Student {

    private List<Course> courseList = new
       ArrayList<>();

    public void addCourse(Course course) {
        this.courseList.add(course);
    }

    public List<Course> getCourseList() {
        return courseList;
    }
}
```

```
Wanted but not invoked:
course.canEnroll(
    de.tum.in.ase.pse.Student@732f29af
);
-> at de.tum.in.ase.pse.EnrollmentServiceTest.testEnrollStudentSuccessful(EnrollmentServiceTest.java:27)
Actually, there were zero interactions with this mock.

    at de.tum.in.ase.pse.EnrollmentServiceTest.testEnrollStudentSuccessful(EnrollmentServiceTest.java:27)
```

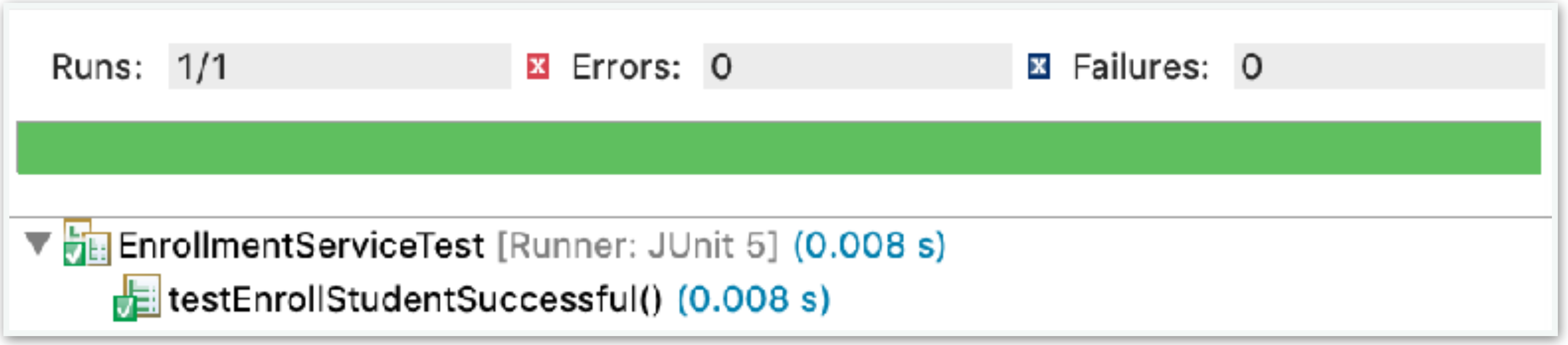
Problem: even if all **assertions** pass, the test fails

Reason: the method **canEnroll** on **Course** was not invoked

Example: correct SUT code leads to a passing test

```java
public class EnrollmentService {

    public boolean enroll(Student student, Course course) {
        if (course.canEnroll(student)) {
            student.addCourse(course);
            return true;
        }
        return false;
    }
}
```

```java
public class Student {

    private List<Course> courseList = new
       ArrayList<>();

    public void addCourse(Course course) {
        this.courseList.add(course);
    }

    public List<Course> getCourseList() {
        return courseList;
    }
}
```



```
Runs: 1/1    ☒ Errors: 0    ☒ Failures: 0

▼ EnrollmentServiceTest [Runner: JUnit 5] (0.008 s)
      testEnrollStudentSuccessful() (0.008 s)
```

Even if the implementation of `Course` is missing, we can test if the implementation of `EnrollmentService` is correct

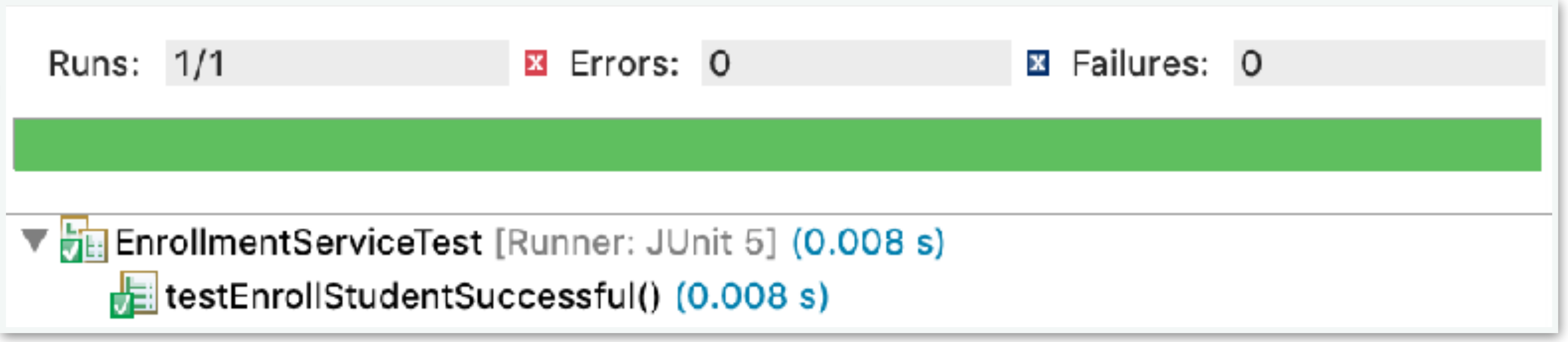
Example: verify void methods

```java
@ExtendWith(MockitoExtension.class)
class EnrollmentServiceTest {

    private EnrollmentService enrollmentService = new EnrollmentService();

    private Course courseMock = mock(Course.class);

    @Test
    void testEnrollStudentSuccessful() {

        Student student = new Student();
        int expectedSize = student.getCourseList().size() + 1;

        when(courseMock.canEnroll(student)).thenReturn(true);

        enrollmentService.enroll(student, courseMock);

        assertEquals(expectedSize, student.getCourseList().size());
        assertTrue(student.getCourseList().contains(courseMock));

        verify(courseMock).canEnroll(student);
        verify(courseMock).start();
    }
}
```

```java
public class Student {

    private List<Course> courseList = new
       ArrayList<>();

    public void addCourse(Course course) {
        this.courseList.add(course);
        course.start();
    }

    public List<Course> getCourseList() {
        return courseList;
    }
}
```

No need to specify the behavior of void methods

The order of the method calls is not relevant in this test, but can be verified with an **`inOrder`** verifier



```
Runs: 1/1    Errors: 0    Failures: 0

▼ EnrollmentServiceTest [Runner: JUnit 5] (0.008 s)
    testEnrollStudentSuccessful() (0.008 s)
```

Mock vs. spy

1. **`mock()/@Mock`:** full mocking
   - Optionally specify how it should behave via `Answer/MockSettings`
   - **`when()/given()`** to specify how a mock should behave
   - If the provided answers don't fit your needs, write one yourself extending the `Answer` interface
2. **`spy()/@Spy:`** partial mocking
   - The behavior of single methods can be specified
   - Real methods are invoked but still can be verified and stubbed

Argument matchers

- Allow more flexible verification or mocking, e.g. any integer
- If you are using argument matchers, all arguments have to be provided by matchers
- Examples

```
verify(mock).someMethod(anyInt(), anyString(), eq("third argument"));
//above is correct - eq() is also an argument matcher

verify(mock).someMethod(anyInt(), anyString(), "third argument");
//above is incorrect - exception will be thrown because third argument is given without an argument matcher

when(mock).anotherMethod(anyInt(), anyString()).thenReturn(true);
//if you want all method calls to return true
```

Does not work, argument matchers cannot be mixed with actual arguments

Inject mocks into SUT code

- **`@InjectMocks:`** injects mock or spy fields into test objects automatically
- Mockito will try to instantiate objects annotated with **`@Spy`** and **`@Mock`** and will instantiate **`@InjectMocks`** fields using constructor injection, setter injection, or field injection
- Example

More about dependency injection in **L09 Testing Patterns II**

```java
@ExtendWith(MockitoExtension.class)
class PubTest {

    @Spy
    BeerDrinker drinker;

    @InjectMocks
    LocalPub localPub;

    @Test
    void testHappyHour() {
        localPub.happyHour();
        verify(drinker).drinkBeer();
    }
}
```

At the start of the test, a **BeerDrinker** object is automatically created and use to create a **LocalPub** object using **constructor injection**

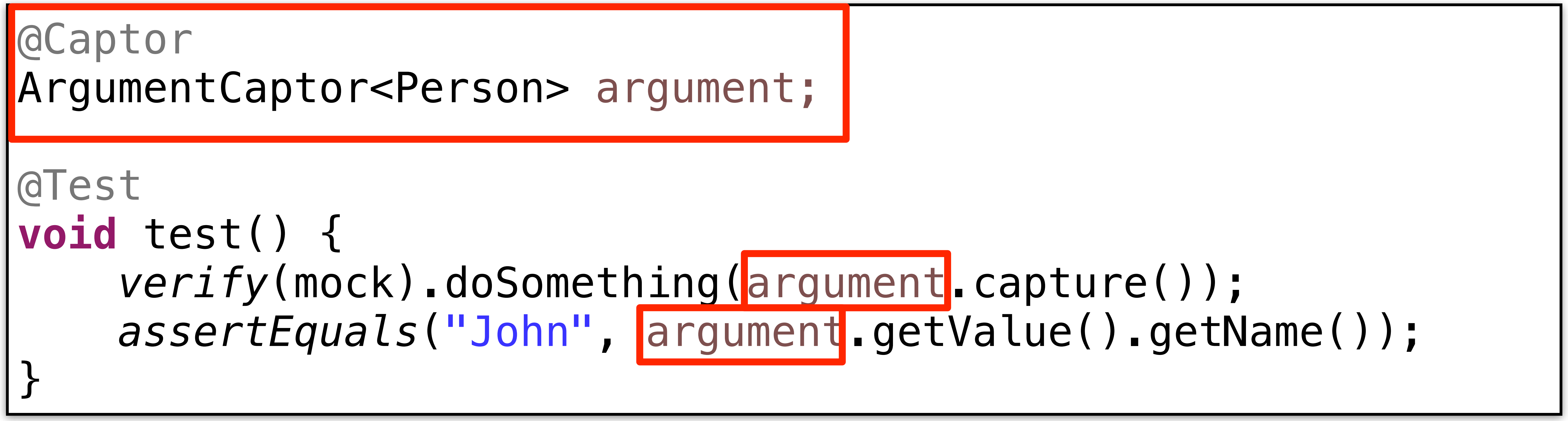
```java
public class LocalPub {
    private BeerDrinker beerDrinker;

    public LocalPub(BeerDrinker beerDrinker) {
        this.beerDrinker = beerDrinker;
    }

    public void happyHour() {
        beerDrinker.drinkBeer();
    }
}
```

Constructor used to inject the **BeerDrinker** object automatically

```java
public class BeerDrinker {

    public void drinkBeer() {
        System.out.println("Drink Beer");
    }
}
```

Argument captors

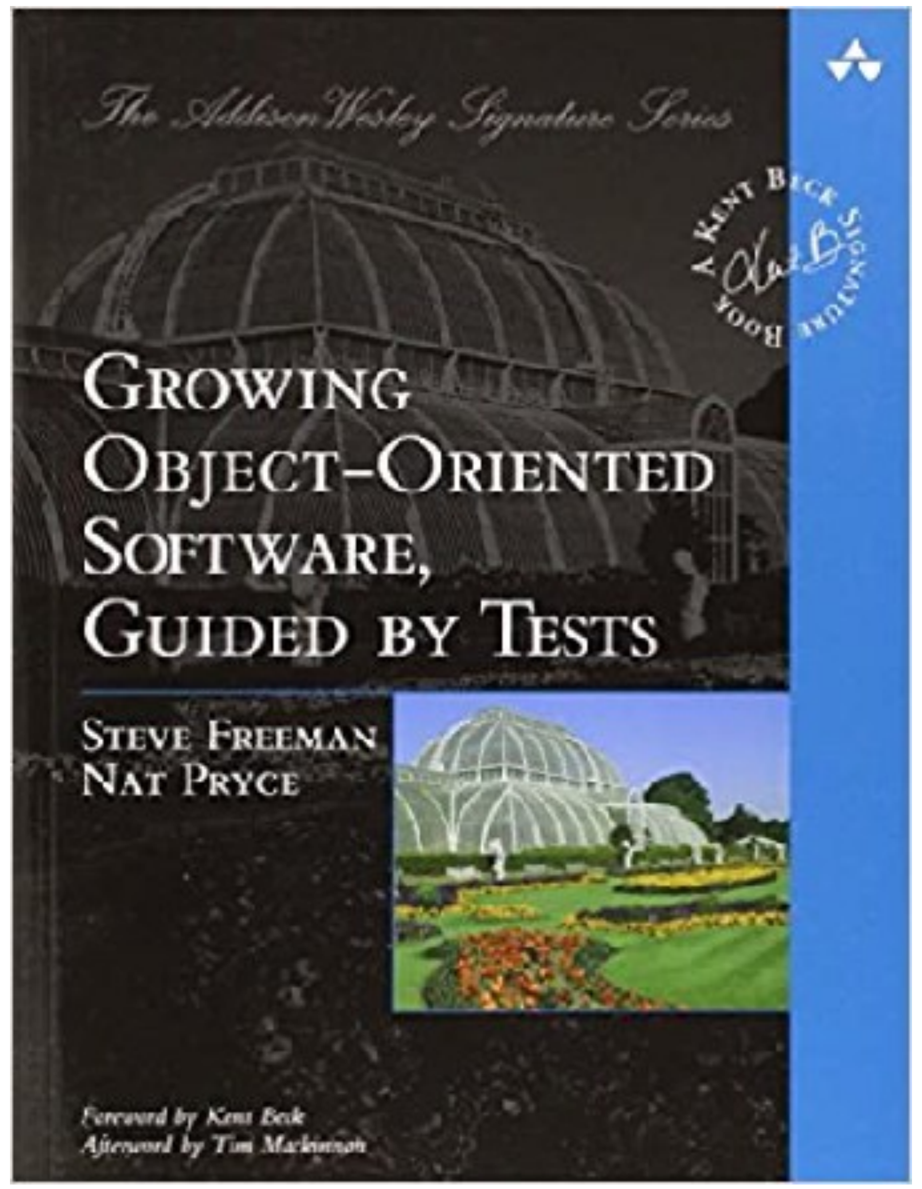
- Capture argument values for further assertions
- Mockito verifies argument values in natural java style: by using an **`equals()`** method
- This is also the recommended way of matching arguments because it makes tests clean and simple
- In some situations though, it is helpful to assert on certain arguments after the actual verification
- Example

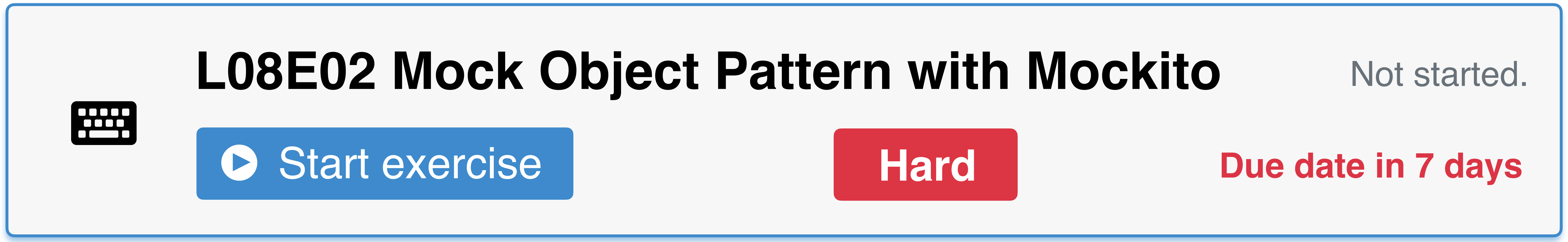


```java
@Captor
ArgumentCaptor<Person> argument;

@Test
void test() {
    verify(mock).doSomething(argument.capture());
    assertEquals("John", argument.getValue().getName());
}
```

How to write good tests

- Keep the testing code compact and readable
- Avoid coding a tautology
- Cover as much of the range as possible to show positive cases and especially erroneous code paths
- Don't mock a type you don't own
- Don't mock everything, it's an antipattern
- Don't mock value objects
- Read "Growing Object Oriented Software Guided by Tests”


The Addison-Wesley Signature Series
A Kent Beck Signature Book
Growing Object-Oriented Software, Guided by Tests
Steve Freeman
Nat Pryce
Foreword by Kent Beck
Afterword by Tim Mackinnon



L08E02 Mock Object Pattern with Mockito

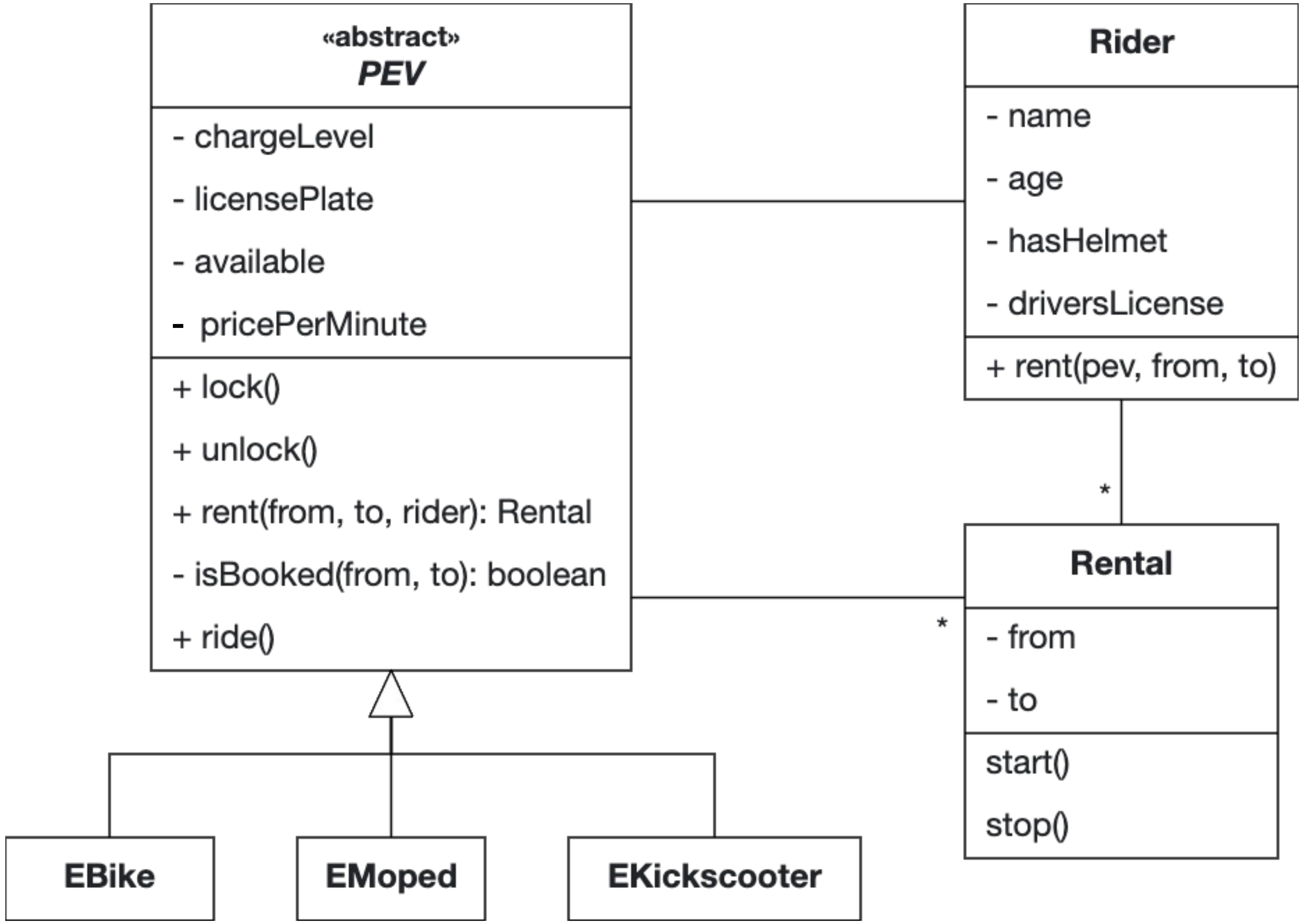
Not started.

Due date in 7 days

20 min

14 pts

- Problem statement: PEV rentals **Reservation System**
- Advanced mocking techniques
  - Spy vs. mock
  - Argument captor
  - Argument matcher
  - Inject mocks



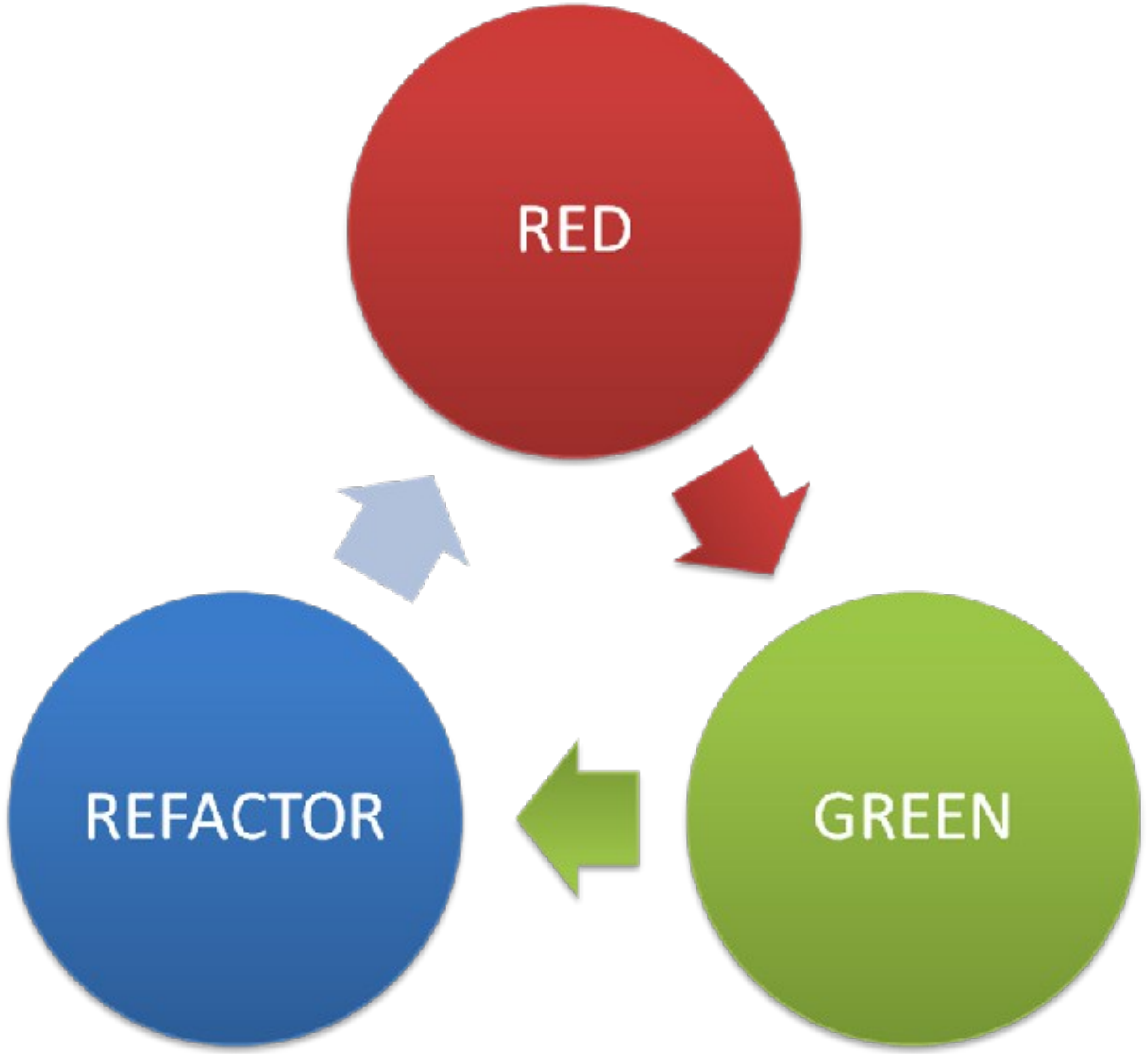
EasyMock vs. Mockito

- For simple cases, both offer similar functionality
  - Specify the desired behavior of the collaborators (stubbing / mocking)
  - Verify that the methods have been called (optionally in a specific order)
- Mockito has a clear separation between specifying the behavior and verifying that certain methods are invoked
- Mockito offers spies, i.e. partial mocks
- EasyMock always requires the **`replay()`** method in order to verify
- EasyMock distinguishes between **`nice`**, **`default`** and **`strict`** mocks
- EasyMock has difficulties with stubbing **`void`** methods
- https://github.com/mockito/mockito/wiki/Mockito-vs-EasyMock

Outline

- Testing
- Mock object pattern
  - EasyMock
  - Mockito
- ➡ Test driven development
- Reflection test pattern
- Four stage testing pattern
- Testing patterns for MVC

The test driven development (TDD) cycle



1. Write a test

- Before writing any other code

2. Get the test to pass

- With the most basic solution

3. Optimize the design

- Make code more readable
- Eliminate “code smells”
- Make it pretty

- **Repeat**

Why use TDD?

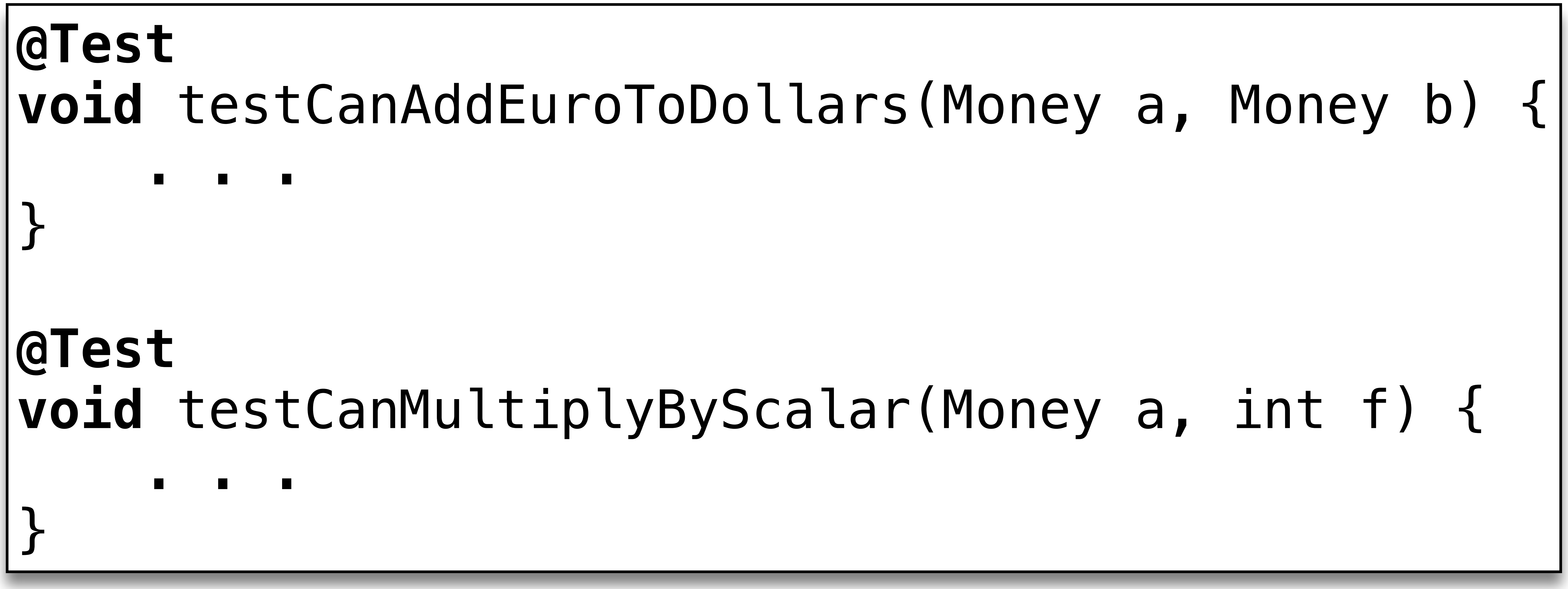
In TDD, tests **drive** the development

```
@Test
void testMoneyMeetsRequirements() {
    // TODO: 5 EUR * 2 = 10 EUR
    // TODO: 5 EUR + 10 GBP = 10 EUR if rate is 2:1
    fail("complete integration test");
}
```

They can be seen as requirements and documentation

Why use TDD?

If your tests are well written…



```
@Test
void testCanAddEuroToDollars(Money a, Money b) {
    . . .
}

@Test
void testCanMultiplyByScalar(Money a, int f) {
    . . .
}
```

… they serve as a concise documentation of what your implementation can and cannot do

Benefits

- Many teams report **significant reductions** in defect rates, at the cost of a moderate increase in initial development effort
- The same teams tend to report that these overheads **significantly reduce the effort** in the final phases of a project (for testing and documentation)
- Practitioners report that **TDD** leads to **improved design qualities** in the code (mainly on the object design level)
- Higher degree of “internal” or technical quality, for instance improving the metrics of cohesion and coupling

Limitations

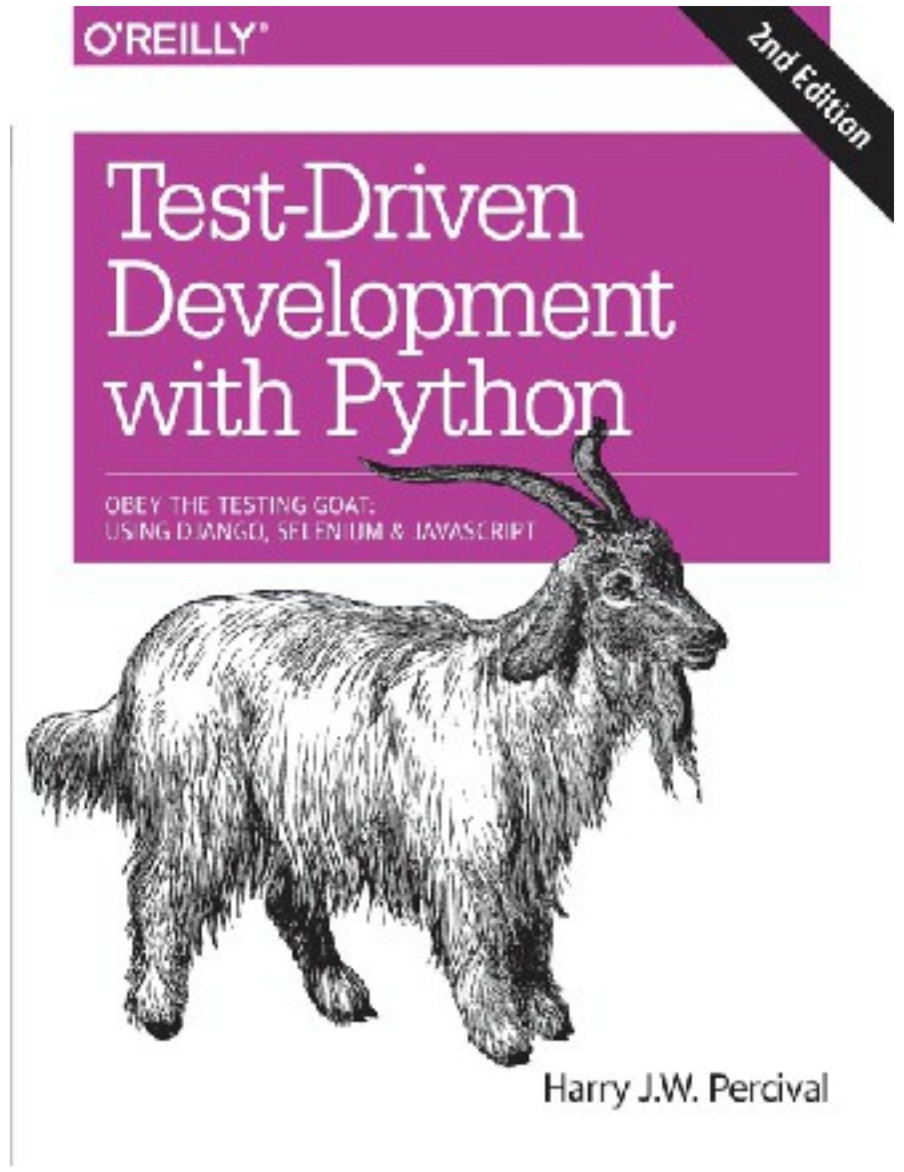
- It can be very tedious in the beginning
  - You need to write tests for everything
  - You need a lot of (small) tests

- It can be frustrating, even (especially?) for experienced developers
  - You need to stick to the TDD cycle and write tests before you write code…
    - … no matter how strong the temptation to skip ahead and just write the code
  - You need to learn how to write good tests, that drive your design…
    - … a lot of intuitive choices you make without really thinking about it, need to be made explicit

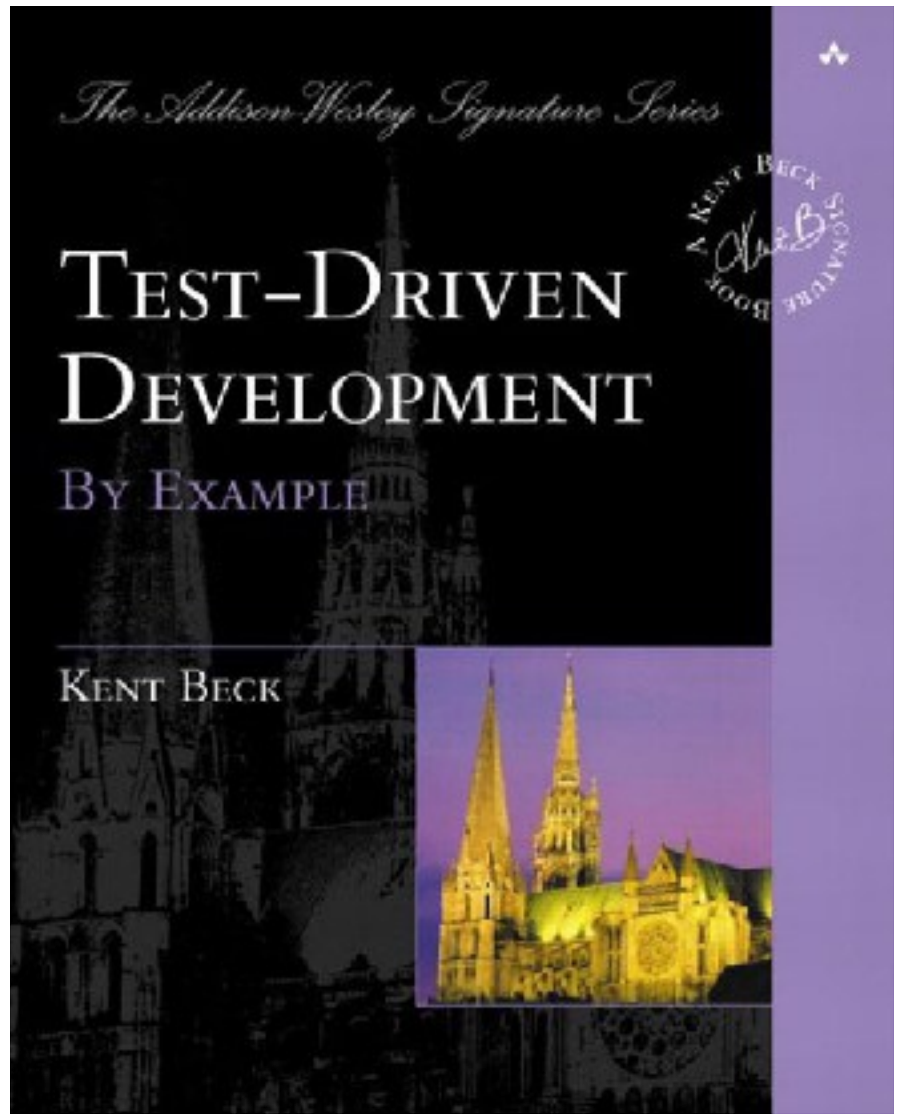
- No system design upfront can lead to issues in the architecture

Literature

- Test Driven Development with Python by Harry Percival


O'REILLY
2nd Edition
Test-Driven Development with Python
OBEY THE TESTING GOAT: USING DJANGO, SELENIUM & JAVASCRIPT
Harry J.W. Percival

- Test Driven Development: By Example by Kent Beck

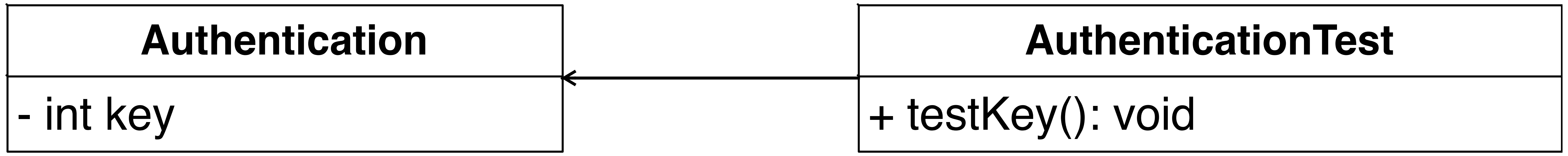

The Addison-Wesley Signature Series
A Kent Beck Signature Book
Test-Driven Development
By Example
Kent Beck

Outline

- Testing
- Mock object pattern
  - EasyMock
  - Mockito
- Test driven development
- ➡ Reflection test pattern
- Four stage testing pattern
- Testing patterns for MVC

Reflection test pattern: testing a private attribute

- Example: assume there is no getter for the private field “key” of the Authentication class but we need to access it for our test



Authentication

- int key

AuthenticationTest

+ testKey(): void

```
class AuthPrivacyTest {
    @Test
    void testKey() throws Exception {
        Authentication auth = new Authentication("privateKey");
        Class<? extends Authentication> cl = auth.getClass();

        // get the reflected object
        Field field = cl.getDeclaredField("key");

        // set accessible true
        field.trySetAccessible();
        assertEquals(field.get(auth), privateKey);
    }
}
```

Get the class object for **Authentication**

Get the field “key”

Set the field to be accessible

Reflection test pattern: testing a private attribute

- **Problem**: there are some cases when it is necessary to test a private attribute
    - Legacy code
    - API of open source library (otherwise problems when updating the API)
    - Bad design: refactor, but before refactoring you need to have a test

- **Solution**: use reflection

L08E03 Reflection Test Pattern

Not started.

▶ Start exercise

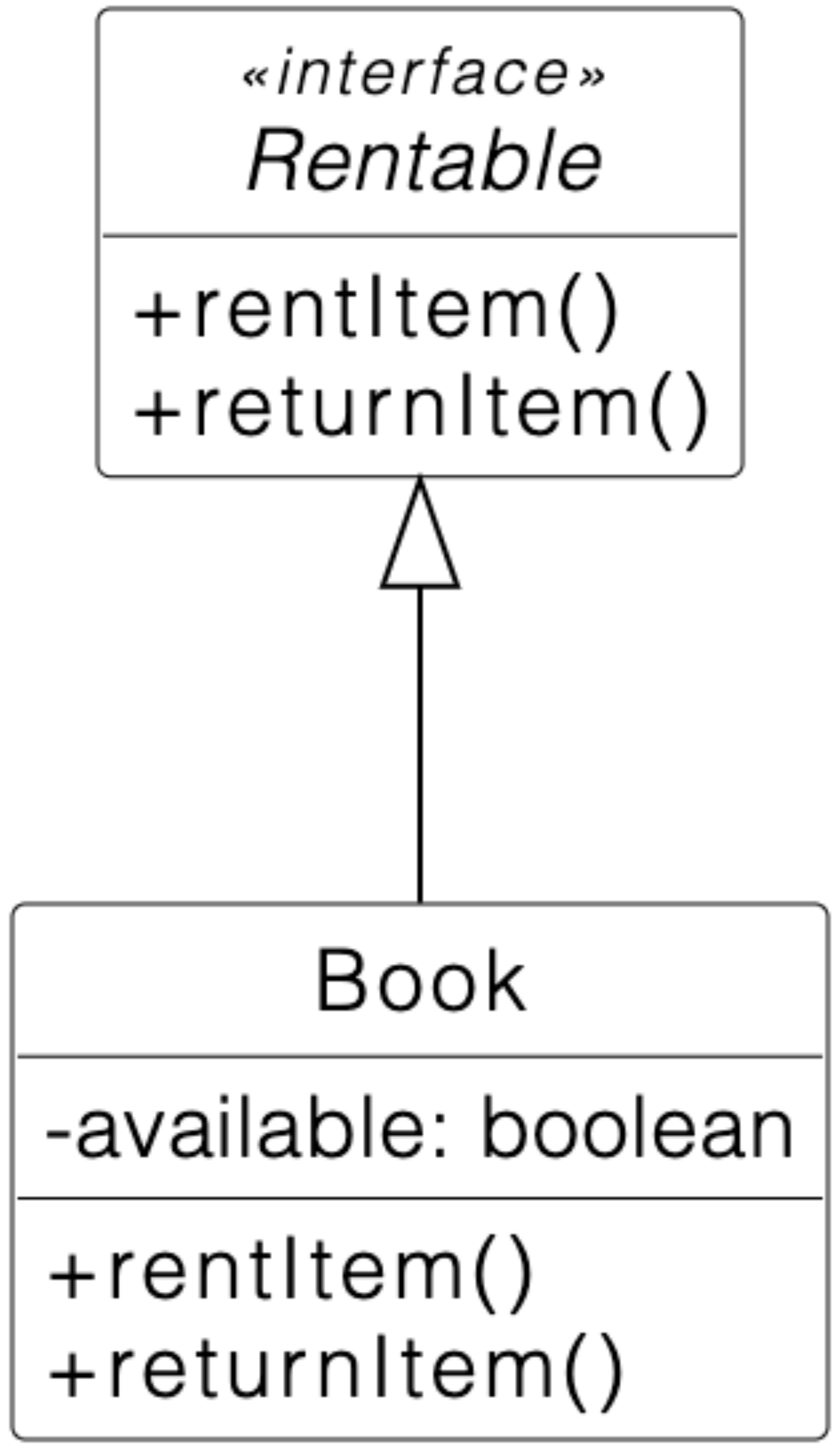
Medium

Due date in 7 days

15 min

8 pts

- Problem statement: **rentable books**
- Assume you are a tutor and need to write an Artemis test
- Test the structure of the "student code" which should follow the given UML class diagram
- The student code might not be available, but you want to provide useful feedback, so you need to use **Reflection**



Reflection

- Versatile way of dynamically linking components
- Manipulation without need to hardcode the target classes

- **Problems**
  - Can obscure what is going on in code
  - Maintenance of reflection code is difficult
  - Results in complex code
  - Performance can be slow

→Use reflection sparingly

→Usually the need to use reflection is a sign for bad design

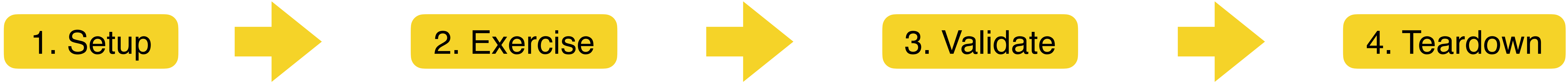
→Refactor after writing the test

Outline

- Testing
- Mock object pattern
  - EasyMock
  - Mockito
- Test driven development
- Reflection test pattern
- ➡ Four stage testing pattern
- Testing patterns for MVC

Four stage testing pattern

- A test driver (actor) executes a flow of events to interact with the SUT



1. Setup → 2. Exercise → 3. Validate → 4. Teardown

1. **Setup:** Create the so-called test fixture with state and behavior that is needed to observe the SUT (such as using a mock object)
2. **Exercise (running the test):** Interact with the SUT, for example by calling a method (often additional setups are required before calling the method)
3. **Validate:** Look at the results with respect to state and/or behavior of the test and determine whether the observed outcome is equal to the expected outcome
4. **Teardown:** Put the SUT back into the state before the test was executed, in particular, tear down any object that was instantiated in the test fixture

Example

```
@ExtendWith(EasyMockExtension.class)
class EnrollmentServiceTest {
    private static EnrollmentService enrollmentService;
    private Student student;

    private Course courseMock = mock(Course.class);

    @BeforeEach
    void setUp() {
        enrollmentService = new EnrollmentService();
    }

    @Test
    void testEnrollmentStudentSuccessful() {
        student = new Student();

        enrollmentService.enroll(student, courseMock);

        assertEquals(...);
        assertTrue(...);
    }

    //...

    @AfterEach
    void tearDown() {
        reset(courseMock);
    }
}
```

1a) Setup collaborating objects

1b) Continued setup:
Create instance of SUT

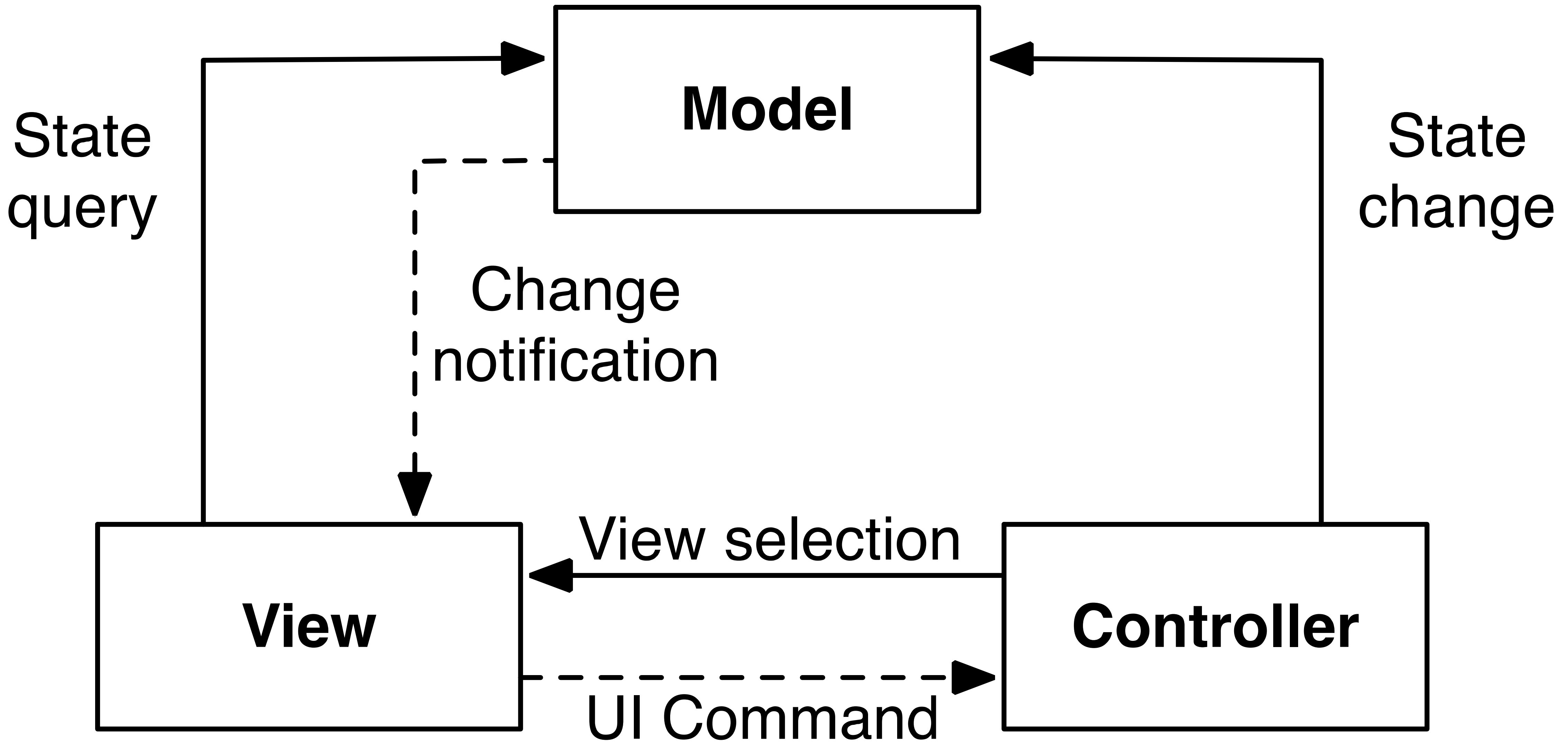
2. Run the test

3. Evaluate the results

4. Tear down the test objects (e.g., reset all mocks)

Two testing patterns for MVC

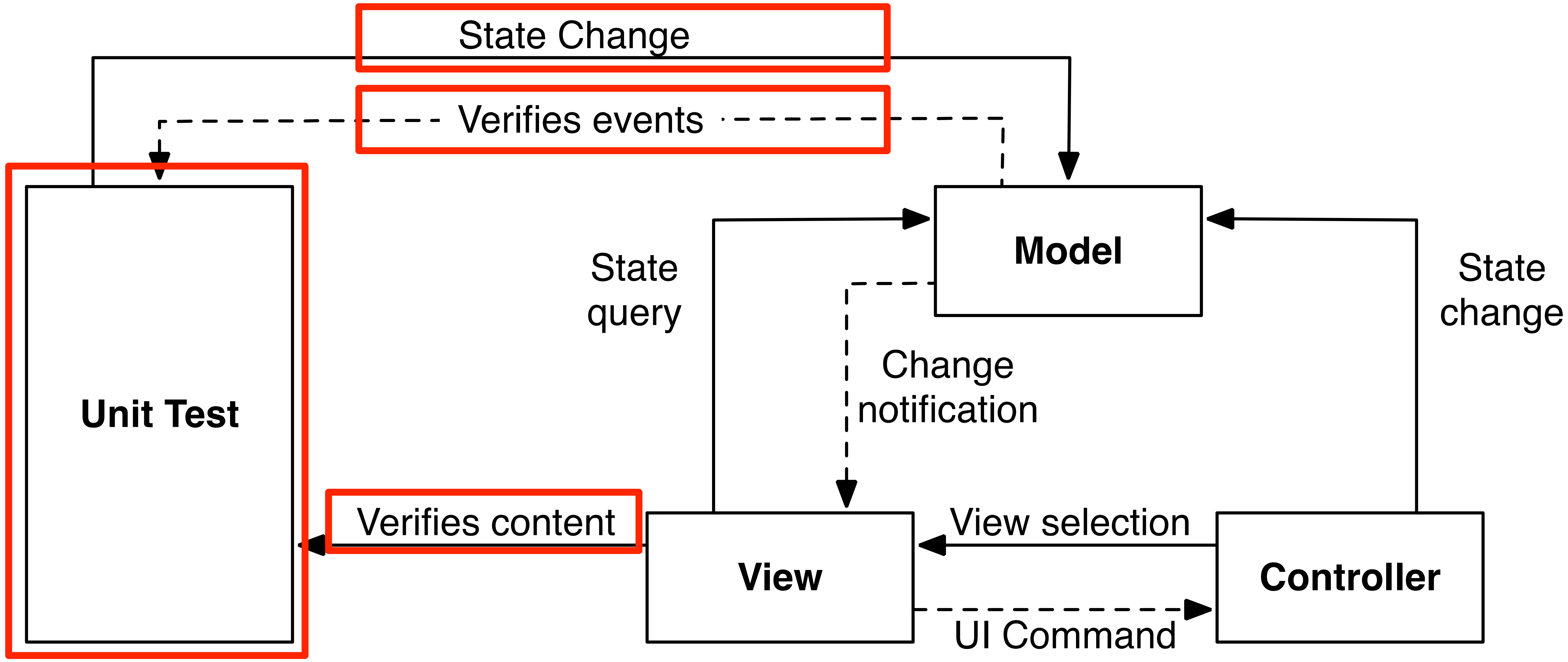
- MVC architectural pattern (review)



- View state test pattern
  - Checks if the view is updated when the model is changed
- Model state test pattern
  - Checks if the model is updated correctly when the user tries to update the model via the view

View state test pattern

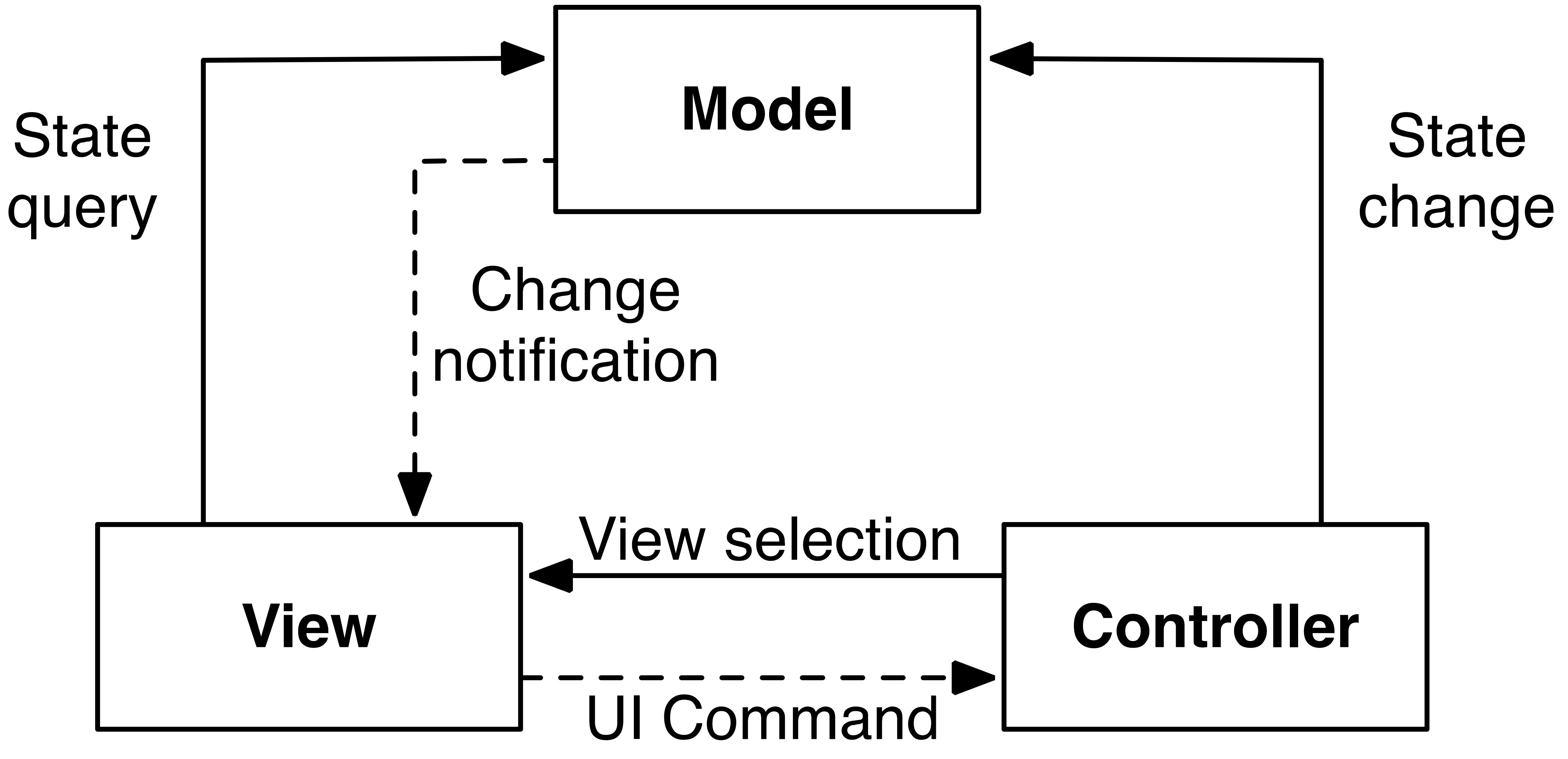
- This pattern tests that whenever the model state changes, the view changes state appropriately



- This test exercises only half of the MVC pattern: the model change notifications to the view, and the view management of those events.
- The controller is not tested in this test pattern

Two testing patterns for MVC

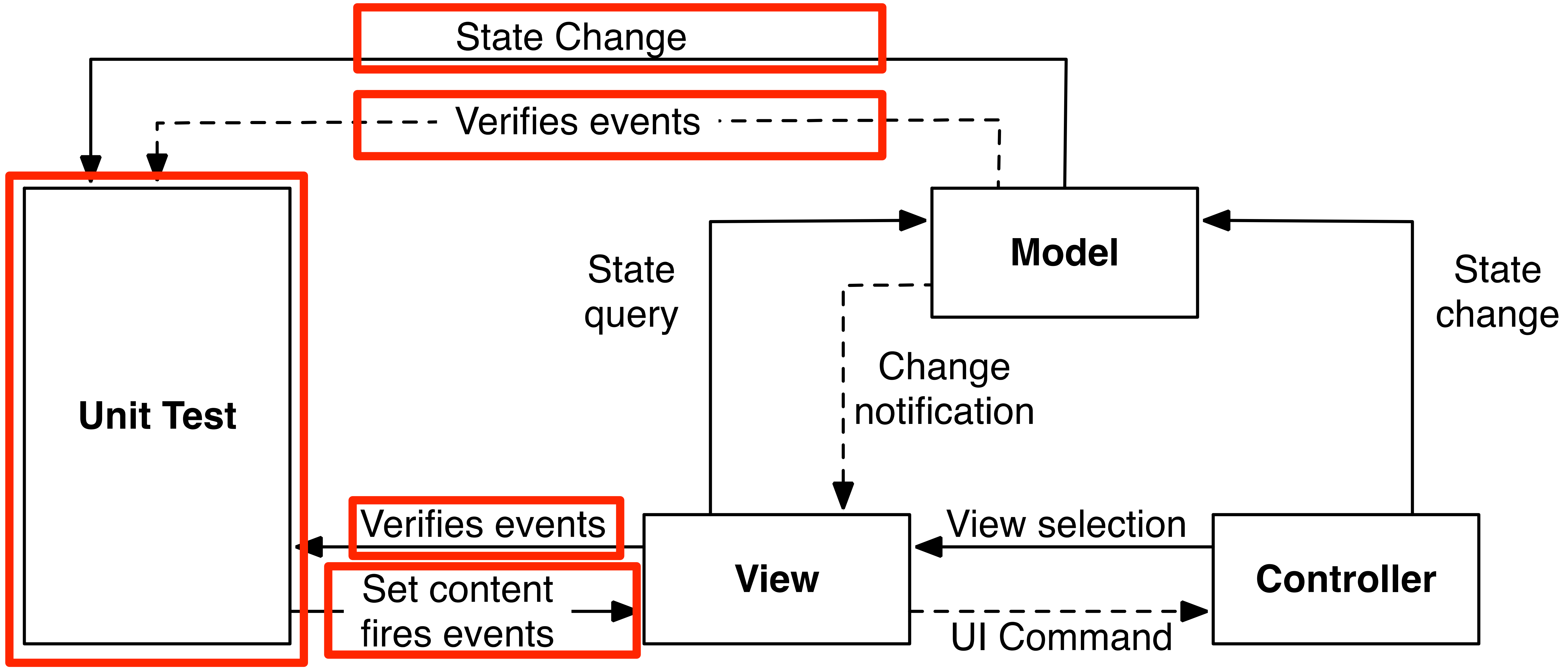
- MVC architectural pattern (review)



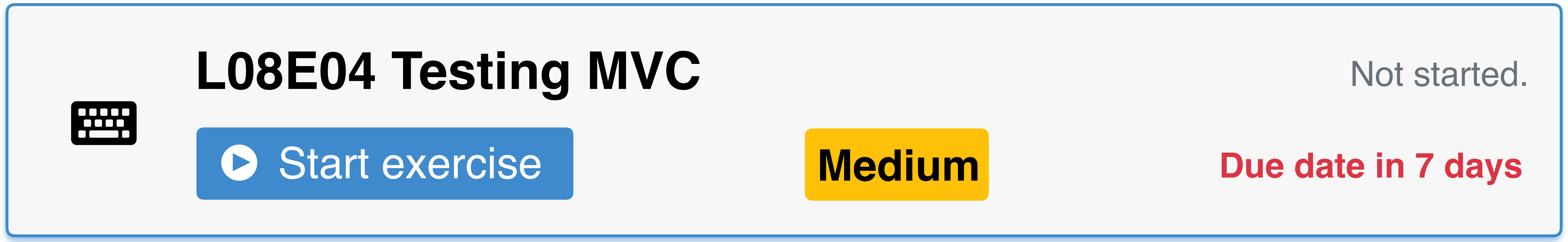
- View state test pattern
  - Checks if the view is updated when the model is changed
- Model state test pattern
  - Checks if the model is updated correctly when the user tries to update the model via the view

Model state test pattern

- This pattern simulates user input by invoking a state change event such as "KeyUp", "Click”, entering a test string “Max Musterman”, etc.



- Validates that the model state is changed and the expected events fired correctly
  - May require some setup on the model itself, treats the controller as a black box
  - Model state can be inspected to determine if the controller is managing the model state correctly



L08E04 Testing MVC

Not started.

Start exercise

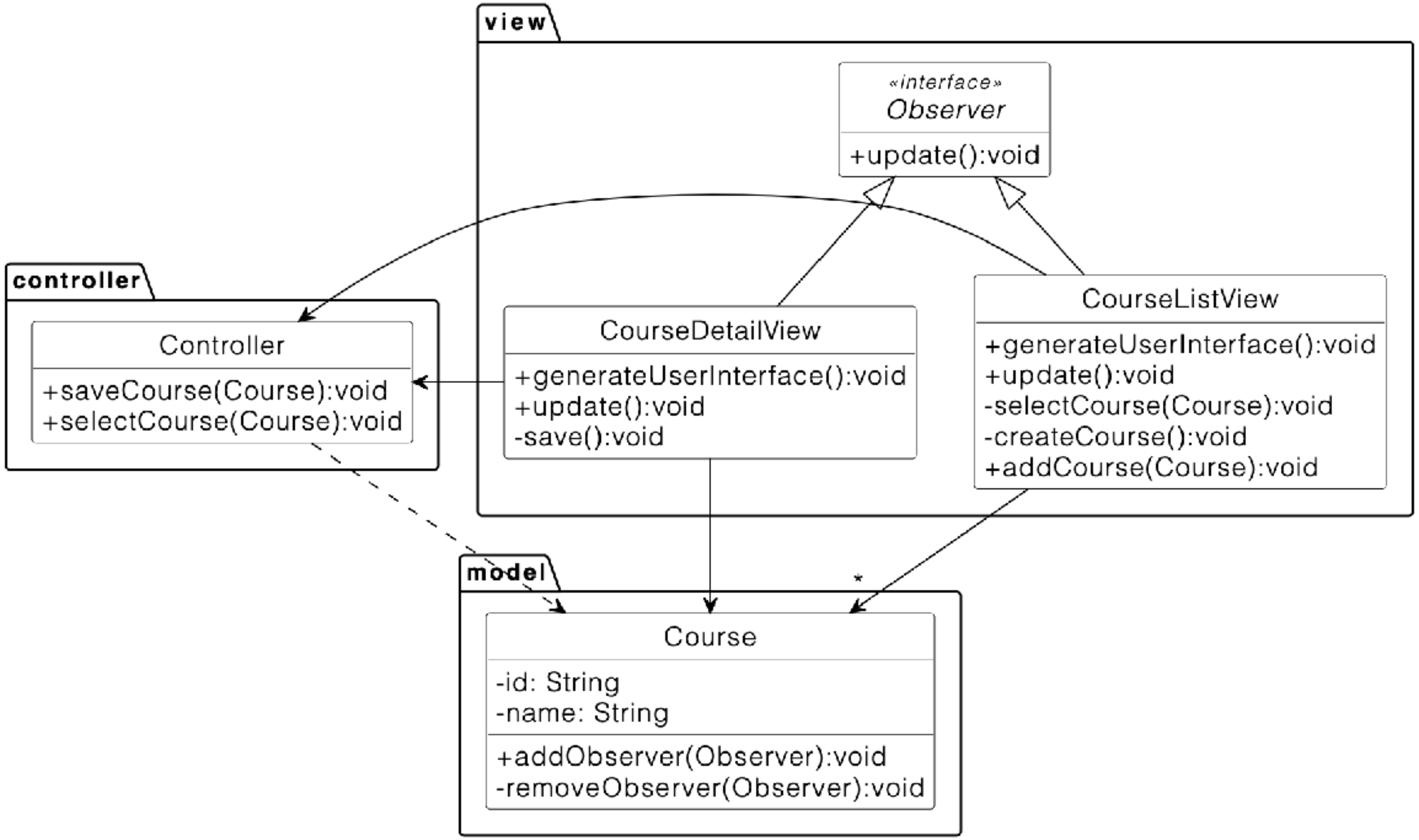
Medium

Due date in 7 days

15 min

8 pts

- Problem statement: test the application developed in **L04E01**



Summary

- Mock objects are test doubles that mimic the behavior of the real object
- The **mock object pattern** enables us to test state and behavior
  - **EasyMock**: simple framework with some limitations
  - **Mockito**: powerful and popular open source framework with many advanced features
- Mock objects are a way to create “self-contained” unit tests (i.e. without much interaction with the rest of the objects in the system model)
- Frameworks for mock objects are available in major programming languages (Java, C++, Perl, Ruby, …)
- **Reflection test pattern:** access or modify private values for testing purposes
- **Testing MVC**: check the view or model state

Literature

- Kent Beck, Erich Gamma, Junit Cookbook: http://junit.sourceforge.net/doc/cookbook/cookbook.htm
- JUnit Source Forge: http://sourceforge.net/projects/junit/files/junit/
- Latest JUnit Source on github: https://github.com/junit-team/
- JUnit Fixtures http://www.informit.com/articles/article.aspx?p=101374&seqNum=5
- Martin Fowler, Mocks are not Stubs: http://martinfowler.com/articles/mocksArentStubs.html
- Brown & Tapolcsanyi: Mock Object Patterns. In Proceedings of the 10th Conference on Pattern Languages of Programs, 2003. http://hillside.net/plop/plop2003/papers.html
- Freeman & Pryce: Growing Object-Oriented Software Guided by Tests: http://www.growing-object-oriented-software.com
- Lars Vogel: Testing with EasyMock: http://www.vogella.com/articles/EasyMock/article.html
- Gerard Meszaros, xUnit Test Patterns, Refactoring Test Code, Pearson Education, 2007
- Jeff Langr , Pragmatic Unit Testing in Java 8 with JUnit, 2015