T06 Software Testing

Prof. Pramod Bhatotia
Systems Research Group
https://dse.in.tum.de/

Tutorial outline

- **Part I: Lecture summary**
  - **Q&A for the lecture material**
- **Part II:** Programming basics
- **Part III:** Homework programming exercises (Artemis)

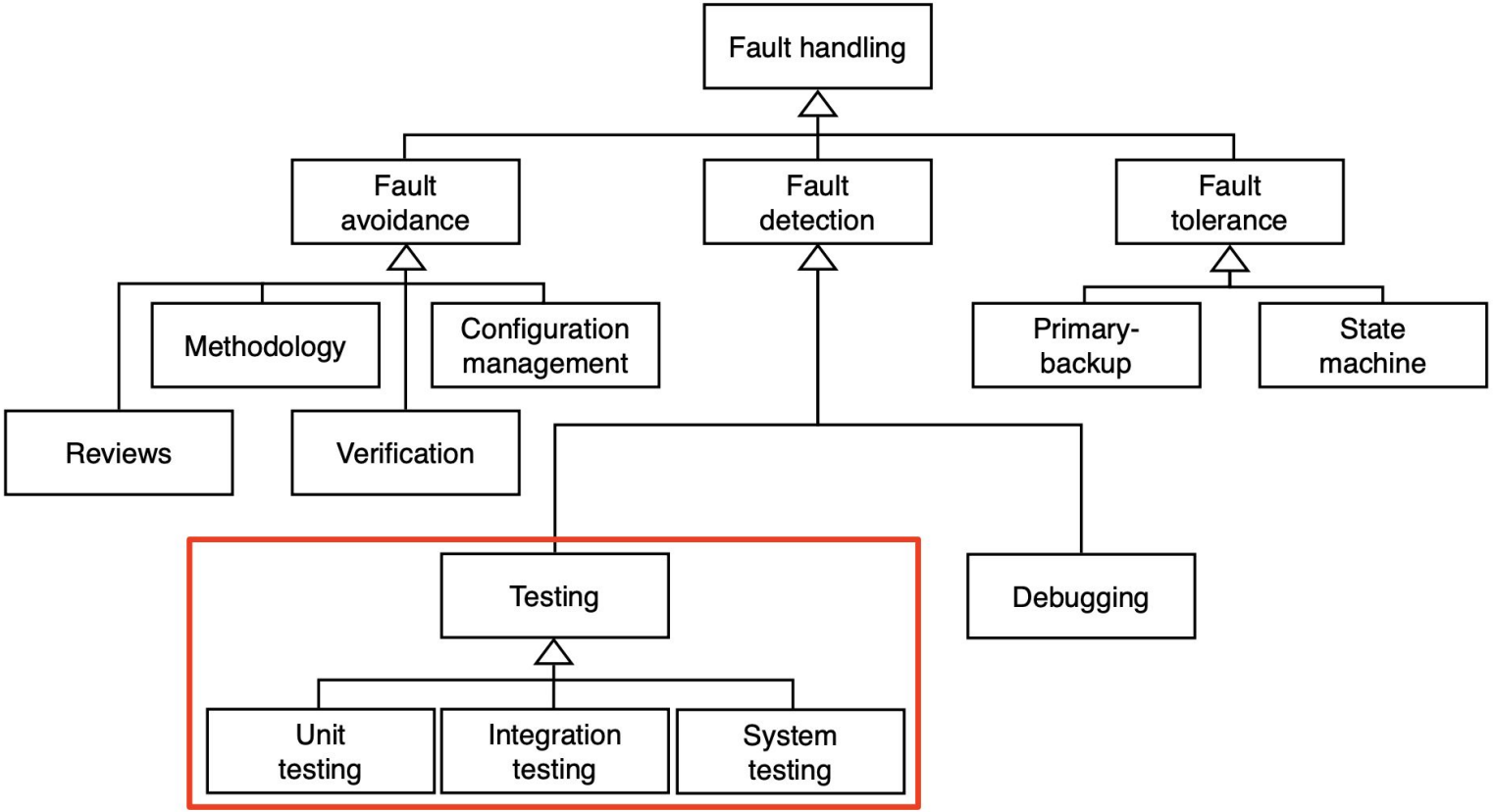
Lecture overview

- **Part I:** Testing activities
  - Terminology
  - Unit testing
  - Integration testing
- **Part II:** Automated system testing
  - Fuzzing
  - Symbolic execution
  - Chaos Monkey
- **Part III & IV:** Model-based and object-oriented testing
  - Mock object pattern

Terminology

- **Failure:** any deviation of the observed behavior from the specified behavior
  - Informally also called **crash**
- **Error:** the system is in a state so that further processing can lead to a failure
  - Also called **erroneous state**
- **Fault:** the mechanical or algorithmic cause of an error
  - Informally also called **bug**
- **Validation:** activity of checking for deviations between the observed behavior of a system and its specified behavior

Taxonomy for fault handling techniques



Types of testing

Unit testing

- The development team tests individual components (method, class, subsystem)
- **Goal:** confirm the component is correct and carries out the intended functionality

System testing

- The development team tests the entire system
  - **Functional** testing validates functional requirements
  - **Structure** testing validates the subsystem decomposition
  - **Performance** testing validates non-functional requirements
- **Goal:** determine if the system meets the requirements (functional and non-functional)

Integration testing

- The development team tests groups of subsystems (eventually the entire system)
- **Goal:** test the interfaces of the subsystems

Acceptance testing

- The client evaluates the system delivered by developers (in the target environment)
- **Goal:** demonstrate that the system meets the requirements and is ready to use

Unit testing

- A testing method where individual **units** in a program are tested
  - Procedural programming: function or procedure
  - Object-oriented programming: **class**
    - A unit can also be an **attribute**, an individual **method** or the **interface** of the class
- Unit tests are short code fragments created by developers or testers during the development process
- Unit tests form the basis of integration testing

Comparison of white and black box testing

White Box Testing

- Potentially infinite number of paths
- Often tests what is done, instead of what should be done
- Cannot detect missing use cases

Black Box Testing

- Potential combinatorial explosion of test cases (valid & invalid data)
- Does not discover extraneous use cases ("features")

→ Both types of testing are needed

Any test is in between white and black box testing and depends on the following

- Number of possible logical paths
- Nature of input data
- Amount of computation
- Complexity of algorithms and data structures

Integration testing

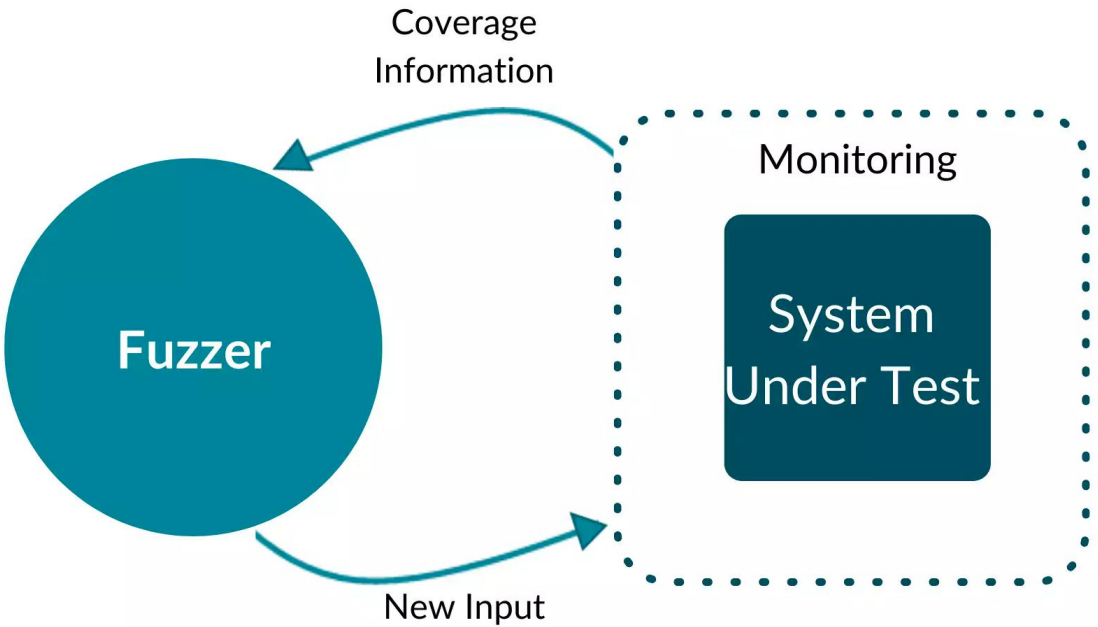
- The entire system is viewed as a collection of subsystems (set of classes) determined during the system and object design
- **Goal:** test all interfaces between subsystems and the interaction of subsystems
- The **integration testing strategy** determines the order in which the subsystems are selected for testing and integration
  - Big bang integration
  - Bottom up testing
  - Top down testing
  - Vertical integration

} Horizontal integration

Fuzz testing

- **(or simply) fuzzing**
  - Run program on many **random, abnormal** inputs and look for **bad behavior** in the responses
  - Bad behaviors such as crashes or hangs
  - Extensively used to find reliability and security issues
- What are the benefits compared to manual testing?
  - (Semi-)automated way of generating a large set of test cases
  - Program is tested using (ab-)normal inputs

Fuzzing


Coverage
Information
Monitoring
Fuzzer
System
Under Test
New Input

Types of fuzzing

- **Black-box fuzzing:** generates input without any knowledge of the program
  - **Mutation-based:** starts with one or more seed inputs → these seeds are modified to generate new inputs: random mutations are applied to the input
  - **Generation-based:** inputs are generated from scratch → structural specification of the input is provided, new inputs are generated to meet the grammar
  - **Example:** Peach
- **Grey-box fuzzing:** involves program instrumentation to get feedback and steer the fuzzer
  - Program is instrumented at the compile time and an initial input seed corpus is provided
  - Seed input is mutated to generate new inputs
  - Generated inputs that cover new control locations (increasing coverage) are added to the seed input
  - **Examples:** AFL, Libfuzz
- **White-box fuzzing:** based on "symbolic execution" that involves program analysis to systematically exercise all paths in the program
  - **Examples:** KLEE, SAGE, Angr, S2E

Symbolic execution

- Symbolic execution generalizes testing
  - Allows unknown symbolic variables in evaluation
- A **symbolic execution engine** executes a program with "symbolic" inputs instead of running the program with regular inputs
  - For e.g., an integer input x is given as value a symbol α that can take on any integer value
- When the program encounters a branch that depends on x, the program state is forked to produce two parallel executions (if and else path)
  - For e.g., make the branch condition evaluate to true (e.g., $\alpha<0$), respectively false (e.g., $\alpha\geq 0$)

Chaos Monkey

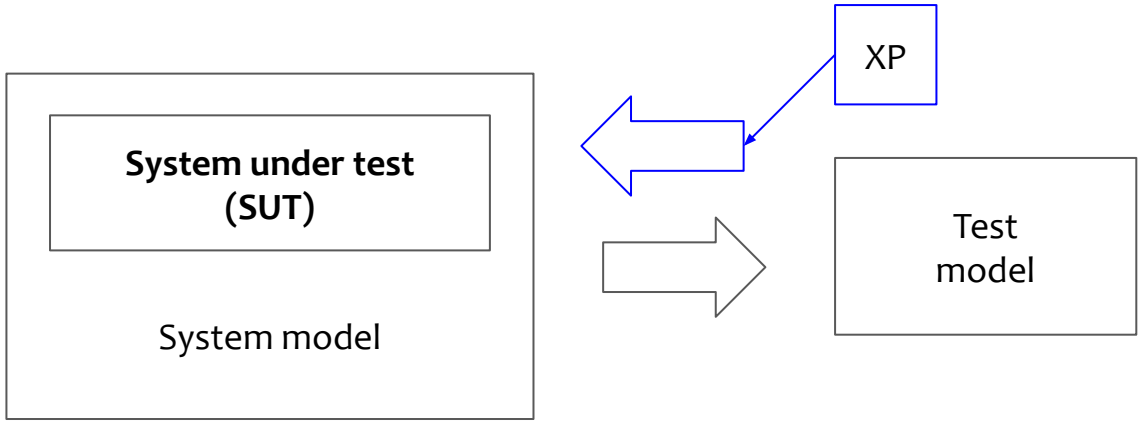
- Chaos Monkey randomly terminates instances in production to ensure that engineers implement their services to be resilient to instance failures
    - A widely used tool, developed at Netflix, to test the resilience of microservices
- How does it work?
    - Set up a cron job that calls Chaos Monkey periodically to create a schedule of terminations
- Tool:
    - https://github.com/Netflix/chaosmonkey

Test model

- Consolidates all test related decisions and components into one package (sometimes also test package or test requirements)
- Contains
  - **Test cases:** functions usually derived from use cases (specification of behavior realizing one or more test objectives)
  - **Input data:** data needed for the execution of the test cases
  - **Oracle:** predicts the expected output data
  - **Test system:** a framework or software component (e.g. JUnit) that executes the tests under varying conditions and monitors the behavior and output

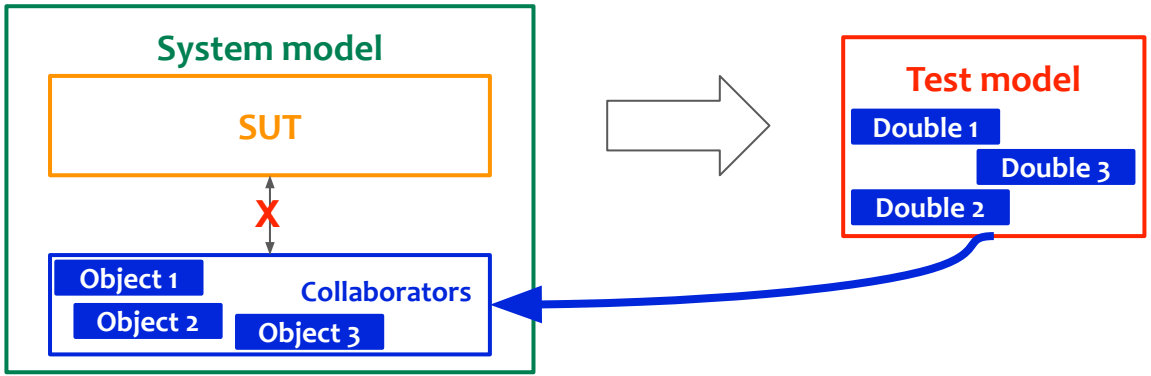
Model based testing

- The **system model** is used for the generation of the **test model**
- Extreme programming **(XP)** variant
  - The **test model** is used for the generation of the **system model**
  - Test-driven development: test → code → refactor
- System under test **(SUT):** part of the system model which is being tested



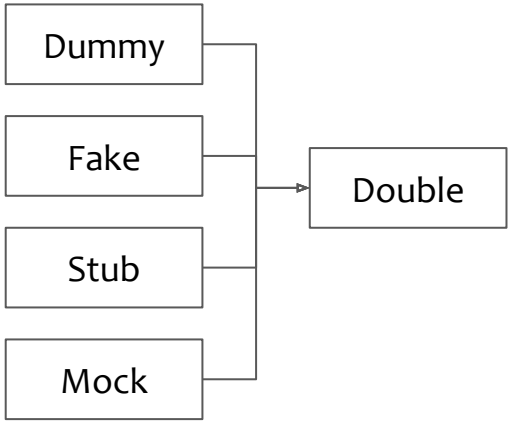
Object oriented test modeling

- 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)



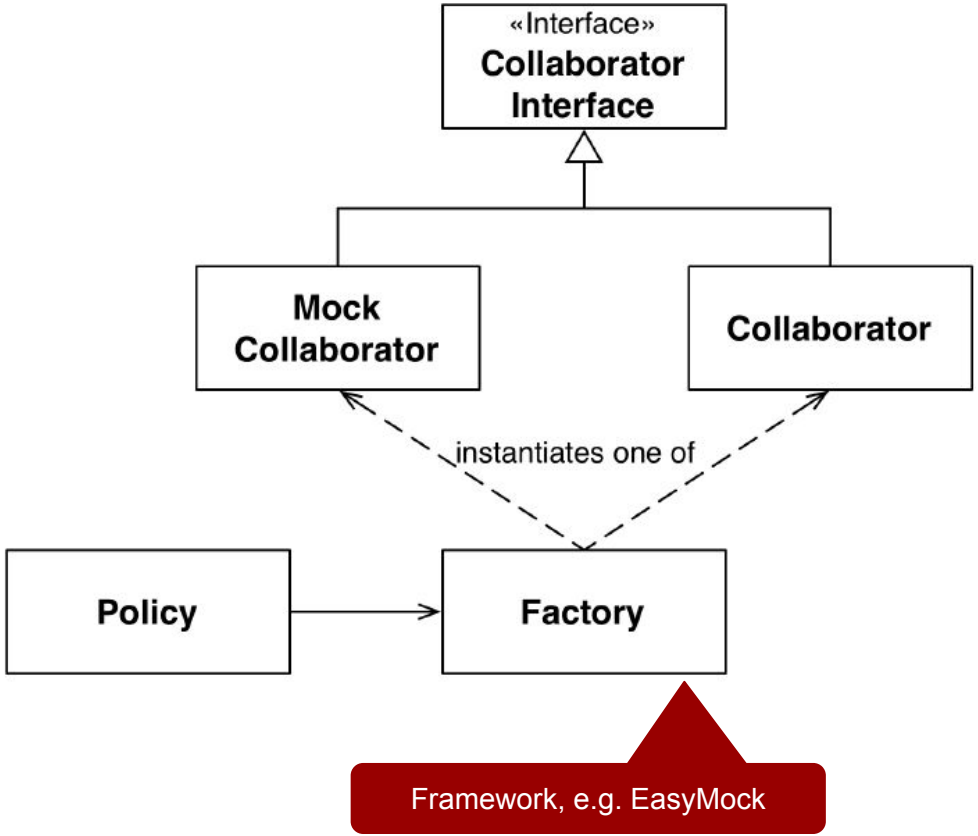
Taxonomy of test doubles

- **Dummy:** often used to fill parameter lists, passed around but never actually used
- **Fake:** a working implementation that contains a "shortcut" which makes it not suitable for production code
  - **Example:** a database stored in memory instead of on a disk
- **Stub:** provides canned answers (e.g. always the same) to calls made during the test
  - **Example:** random number generator that always return 3.14
- **Mock:** mimic the behavior of the real object and know how to deal with a specific sequence of calls they are expected to receive

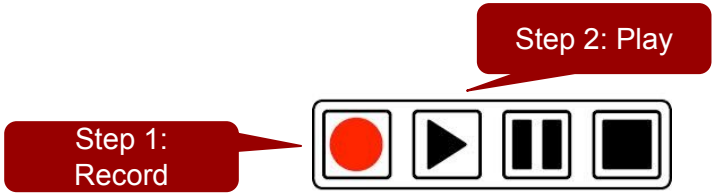


Good design is crucial when using mock objects: the real object (subsystem) must be specified with an interface (façade) and a class for the implementation

Mock object pattern



- A **mock object** replaces the behavior of a real object called the collaborator and returns hard-coded values
- A mock object can be created at startup time with the **factory pattern**
- Mock objects can be used for testing the state of individual objects and the interaction between objects
- The use of mock objects is based on the **record play metaphor**



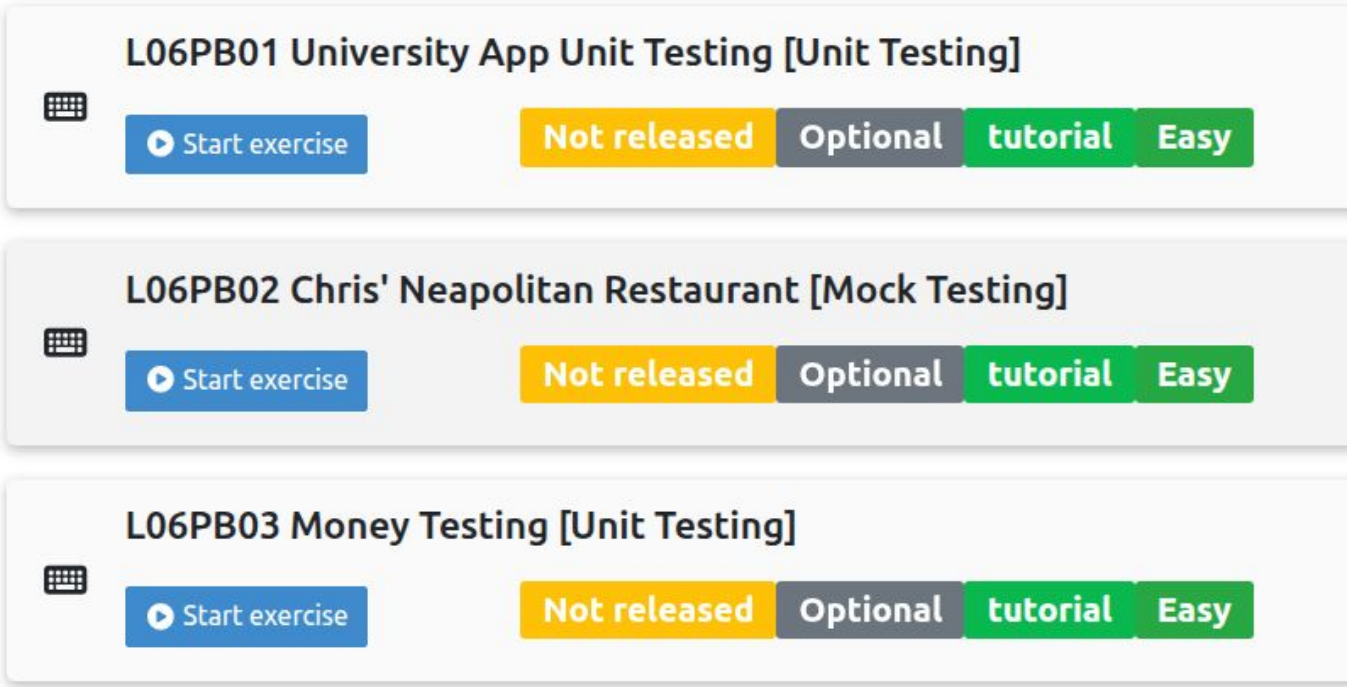
Tutorial outline

- ~~**Part I:** Lecture summary~~
  - ~~Q&A for the lecture material~~

- **Part II:** **Programming basics**

- **Part III:** Homework programming exercises (Artemis)

Programming Basics (PB) exercises



L06PB01 University App Unit Testing [Unit Testing]

- A testing method where individual **units** in a program are tested
  - Procedural programming: function or procedure
  - Object-oriented programming: **class**
    - A unit can also be an **attribute**, an individual **method** or the **interface** of the class
- Unit tests are short code fragments created by developers or testers during the development process
- Unit tests form the basis of integration testing

L06PB01 University App Unit Testing [Unit Testing]

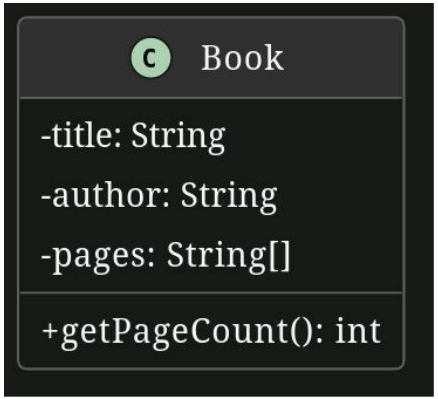
- Tasks:
  - Write unit tests for a module in a TUM University App
  - Implementation in **Java** with **JUnit**

- Goals:
  - **Understand** how to incorporate unit testing in projects

L06PB01 University App Unit Testing [Unit Testing]

To-do:

- Implement four unit tests for the *Book* class
- Test the following methods:
  - *getTitle()* - returns the title
  - *getAuthor()* - returns the author
  - *getPageCount()* - returns the number of pages



L06PB01 University App Unit Testing [Unit Testing]

- Implement the *BookTest* class with JUnit

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

class BookTest {
    private final String testTitle = "Software Engineering Essentials";
    private final String testAuthor = "Book Author";
    private final String[] testPages = {"page1", "page2", "page3"};

    @Test
    void testGetTitle() {
        Book book = new Book(this.testTitle, this.testAuthor,
            this.testPages);
        assertEquals(this.testTitle, book.getTitle());
    }

    @Test
    void testGetAuthor() {
        Book book = new Book(this.testTitle, this.testAuthor,
            this.testPages);
        assertEquals(this.testAuthor, book.getAuthor());
    }
    …
}
```

```java
@Test
void testNoPages() {
    Book book = new Book(this.testTitle, this.testAuthor,
        new String[0]);
    assertEquals(0, book.getPageCount());
}

@Test
void testThreePages() {
    Book book = new Book(this.testTitle, this.testAuthor,
        this.testPages);
    assertEquals(3, book.getPageCount());
}
```

L06PB02 Chris' Neapolitan Restaurant [Mock Testing]

- Tasks:
  - Test the quality of a pizza restaurant service
  - Use mock objects to enable isolated testing of modules
  - Implementation in Java / EasyMock

- Goals:
  - **Understand** how to use mocks in testing
  - **Experience** the isolation mock objects provide when testing individual modules

L06PB02 Chris' Neapolitan Restaurant [Mock Testing]

To-do:

- Write two quality assurance tests
- *testThatTakeawayPizzasAreBoxed():*
  - Makes sure that pizzas for delivery get boxed in
- *testThatTheCorrectTypeOfPizzaIsCreated():*
  - Ensures that the correct type of pizza gets baked
- *infiltrateAniruddhsRestaurant():*
  - Mock an unqualified employee inside the restaurant

L06PB02 Chris' Neapolitan Restaurant [Mock Testing]

- 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

L06PB02 Chris' Neapolitan Restaurant [Mock Testing]

- Implement the quality assurance tests in *PizzaHeavenTest* class

```java
@Test
public void testThatTheCorrectTypeOfPizzaIsCreated() {
    final var orderedPizza = restaurant.orderPizza(
        "Margherita", true);

    assertEquals("Margherita Pizza", orderedPizza.getName());
    assertThrows(NullPointerException.class, () ->
        restaurant.orderPizza("Hawaii", false));
}

@Test
public void testThatTakeawayPizzasAreBoxed() {
    final var pizza = restaurant.orderPizza("Bufalina", true);
    final var unboxedPizza = restaurant.orderPizza(
        "Bufalina", false);

    assertTrue(pizza.isBoxed());
    assertFalse(unboxedPizza.isBoxed());
}
```

L06PB02 Chris' Neapolitan Restaurant [Mock Testing]

- Mock the unqualified malicious employee in *PizzaHeavenTest* class

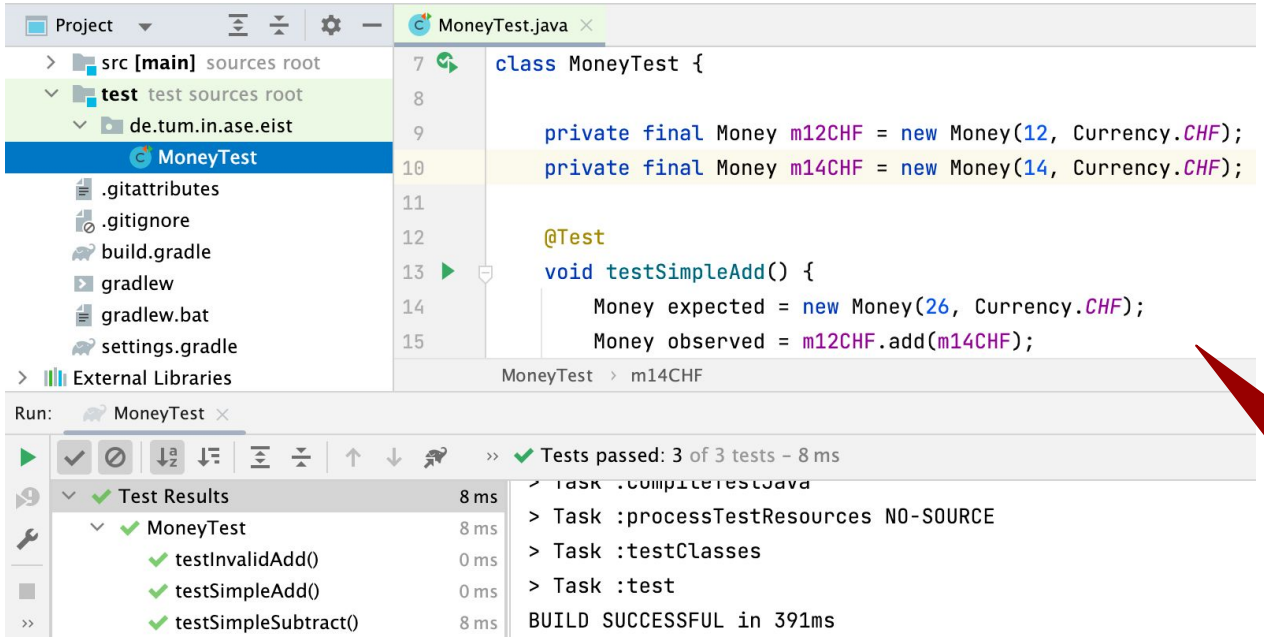
```
@Test
public void infiltrateAniruddhsRestaurant() {
    expect(maliciousEmployee.getName()).andReturn("Aniruddh's Son");

    // prepare the mock here (don't forget to replay it)
    expect(maliciousEmployee.isQualified()).andReturn(true);
    replay(maliciousEmployee);
    this.shopManager = new ShopManager(List.of(maliciousEmployee));

    // test for yourself that the mock works as expected
    // by observing the output of this method
    shopManager.testCurry();

    verify(maliciousEmployee);
}
```

L06PB03: Money Testing [Unit testing]

- Tasks:
  - Execute the test case **MoneyTest**
  - Complete the implementation of **Money** and **MoneyTest**

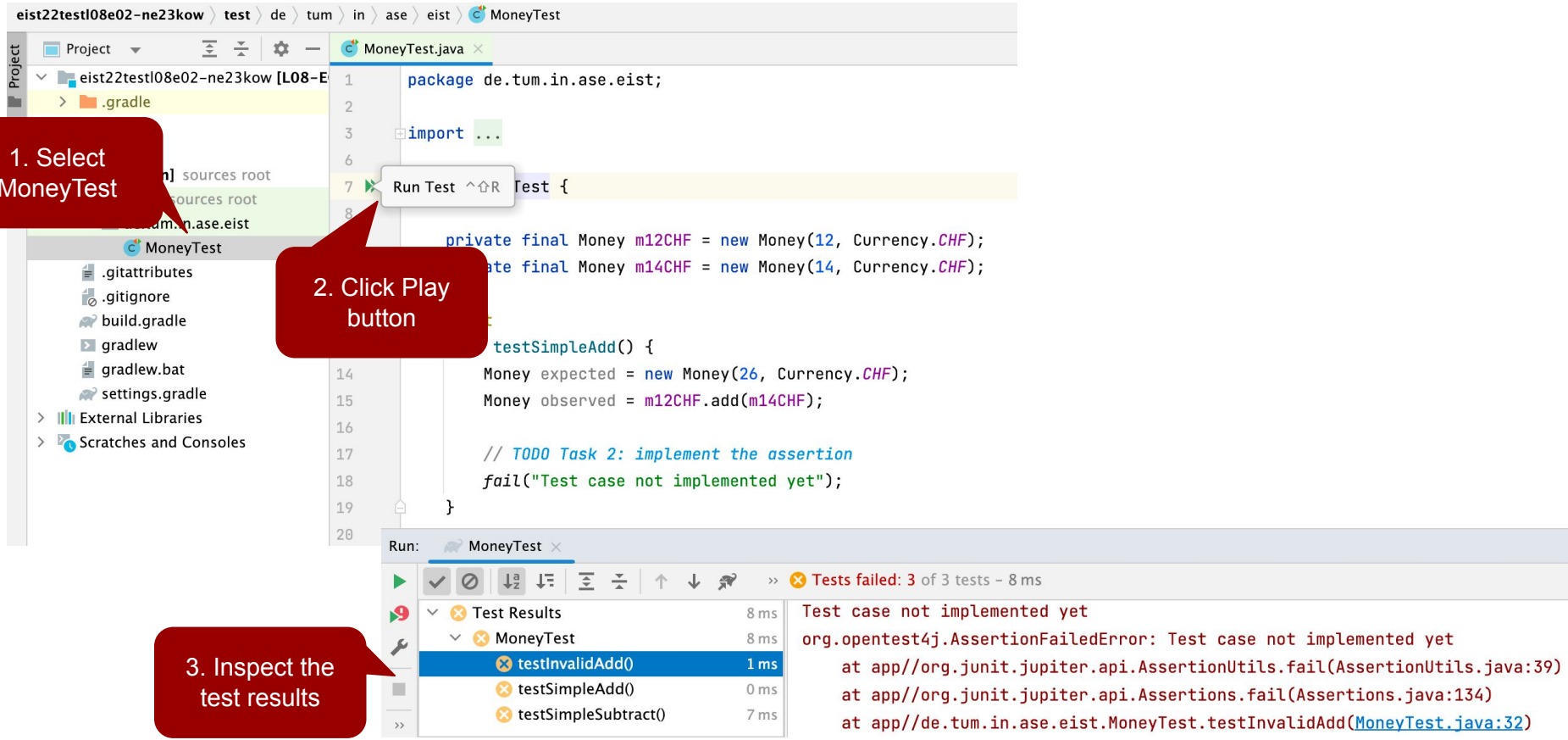
- Goals:
  - Get familiar with a simple unit testing approach

L06PB03: Money Testing [Unit testing]

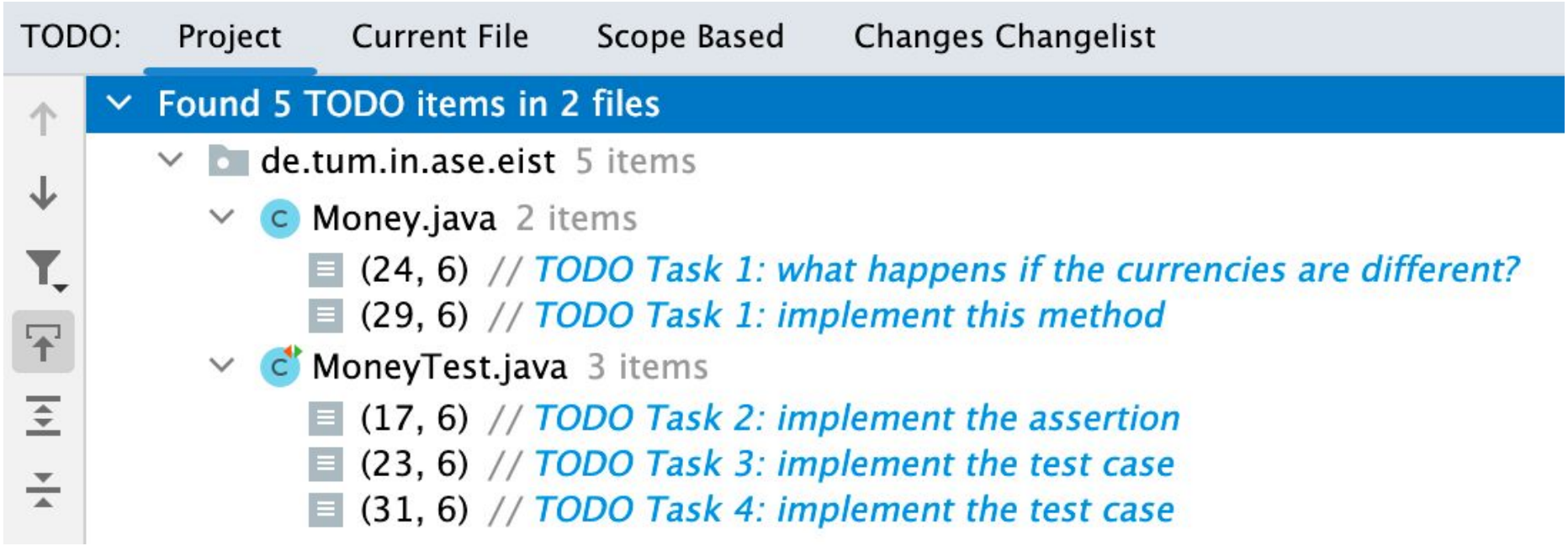


Edsger Dijkstra: Testing shows the presence, not the absence of faults (“bugs”)

L06PB03: Money Testing [Unit testing]



L06PB03: Money Testing [Unit testing]



L06PB03: Money Testing [Unit testing]

```
public class Money {

      //...
      public Money add(Money money) {
            if(currency != money.currency()) {
            throw new IllegalArgumentException("Different currencies not
            supported!");
            }
            return new Money(amount() + money.amount(), currency());
      }

      public Money subtract(Money money) {
            if(currency != money.currency()) {
            throw new IllegalArgumentException("Different currencies not
            supported!");
            }
            return new Money(amount() - money.amount(), currency());
      }

}
```

If the currencies are not the same, throw an IllegalArgumentException

If the currencies are not the same, throw an IllegalArgumentException

L06PB03: Money Testing [Unit testing]

```
class MoneyTest {
      private Money m12CHF = new Money(12, Currency.CHF);
      private Money m14CHF = new Money(14, Currency.CHF);

      @Test
      void testSimpleAdd() {
            Money expected = new Money(26, Currency.CHF);
            Money observed = m14CHF.add(m12CHF);
            assertEquals(expected, observed);
      }

      @Test
      void testSimpleSubtract() {
            Money expected = new Money(2, Currency.CHF);
            Money observed = m14CHF.subtract(m12CHF);
            assertEquals(expected, observed);
      }

      @Test
      void testInvalidAdd() {
            Money m14USD = new Money(14, Currency.USD);
            assertThrows(IllegalArgumentException.class, () -> {
                  m12CHF.add(m14USD);
            });
      }
}
```

Check if the expected amount is the same as the observed amount

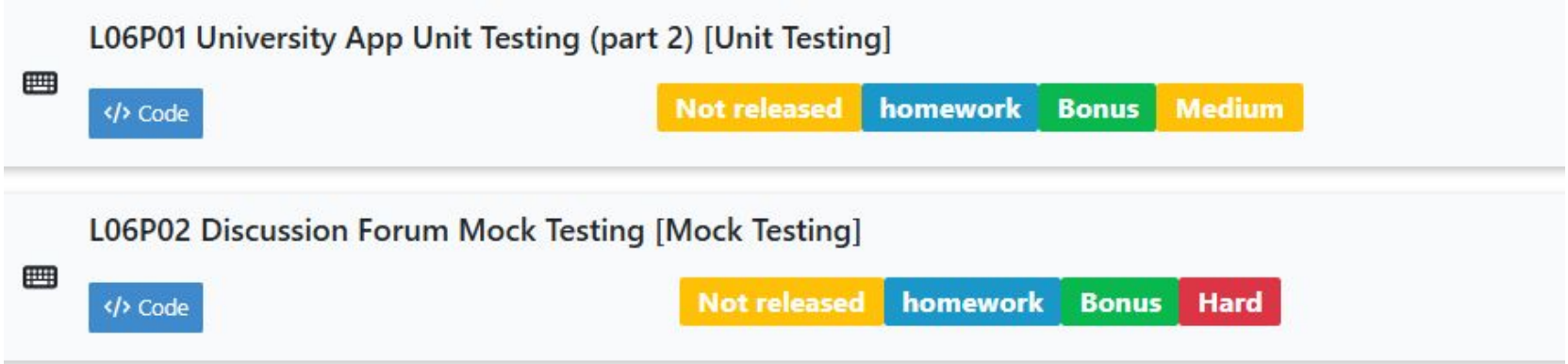
Check if the expected amount is the same as the observed amount

If the currencies are not the same, expect an IllegalArgumentException

Tutorial outline

- ~~**Part I:** Lecture summary~~
  - ~~Q&A for the lecture material~~
- ~~**Part II:** Programming basics~~
- **Part III: Homework programming exercises (Artemis)**

Programming (P) exercises



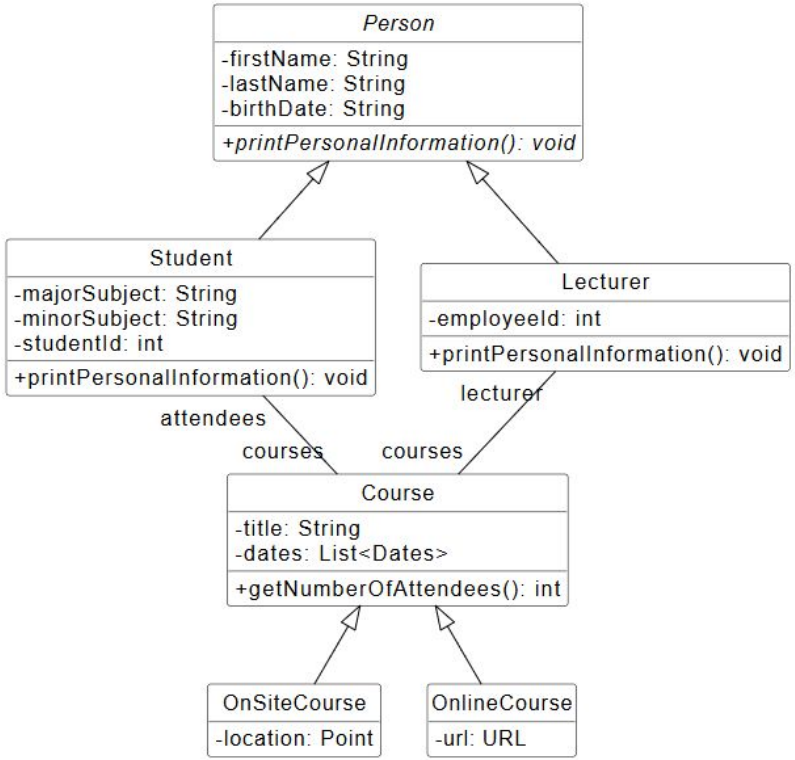
L06P01 University App Unit Testing (part 2) [Unit Testing]

- Tasks:
  - Writing unit tests for a module in a TUM University App
  - Implementation in **Java** with **JUnit**

- Goals:
  - Developing **testing skills** with different scenarios
  - Testing **Exception** and input validation

L06P01 University App Unit Testing (part 2) [Unit Testing]

UML Class diagrams

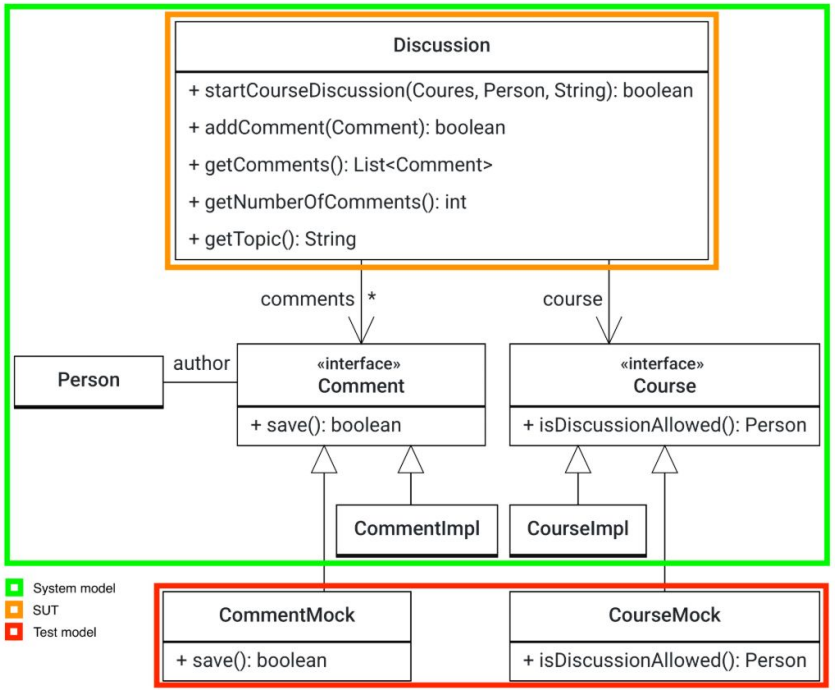


L06P02 Discussion Forum Mock Testing [Mock Testing]

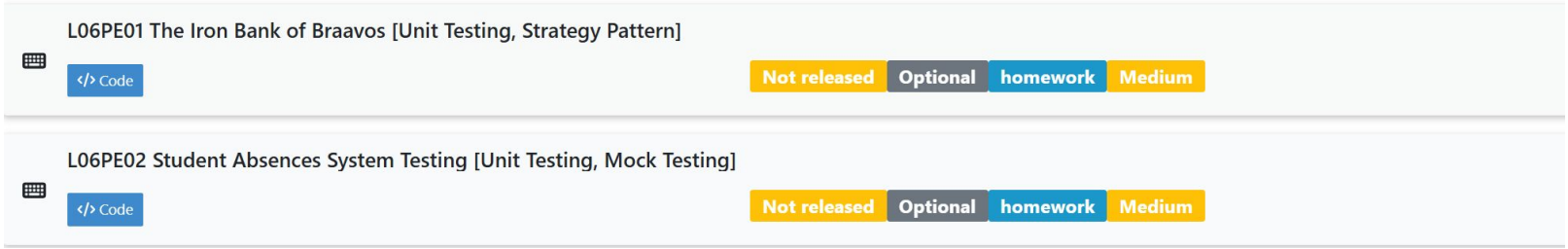
- Tasks:
  - Write unit tests for a class which is dependent on non-fully implemented objects
  - Use the testing framework EasyMock to mock and specify the behaviour of these collaborating objects
- Goals:
  - Practice mocking dependencies
  - Understand how the Mock Object Pattern works

L06P02 Discussion Forum Mock Testing [Mock Testing]

UML diagram of the system model, the SUT, and the test model:



Programming Extras (PE) exercises



L06PE01 The Iron Bank of Braavos [Unit Testing, Strategy Pattern]

- Tasks:
  - **Build** a modular digital banking system
  - **Use** the **Strategy Design Pattern** and **Test-Driven Development (TDD)**
- Goals:
  - **Apply TDD** to a realistic scenario
  - **Practice** writing unit tests
  - **Apply** the Strategy Pattern for flexible and modular design

L06PE02 Student Absences System Testing [Unit Testing, Mock Testing]

- Tasks:
  - Write unit tests to test our student absence system
  - Mock MailService to test email alerts
- Goals:
  - Learn to validate core logic in isolation
  - Practice using mocks to test collaborations
  - You will learn to isolate code under test and mock dependencies

Homework programming exercises

- For more information, please check out the task descriptions on Artemis.

https://artemis.cit.tum.de/