UNIVERSITY OF ICELAND

HBV401G SOFTWARE DEVELOPMENT

10. Software Testing

Matthias Book

Spring 2022

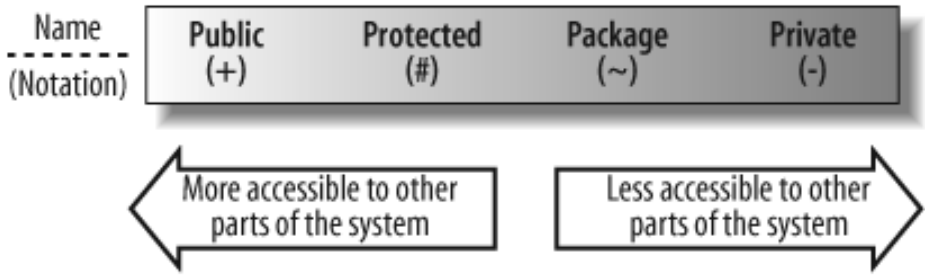
FACULTY OF INDUSTRIAL ENGINEERING, MECHANICAL ENGINEERING AND COMPUTER SCIENCE

Recap: Visibility

Car
– registration – enginePower – wheelSize – fuelSupply – mileage – revolutions – speed
+ accelerate + decelerate + getSpeed + fuelUp

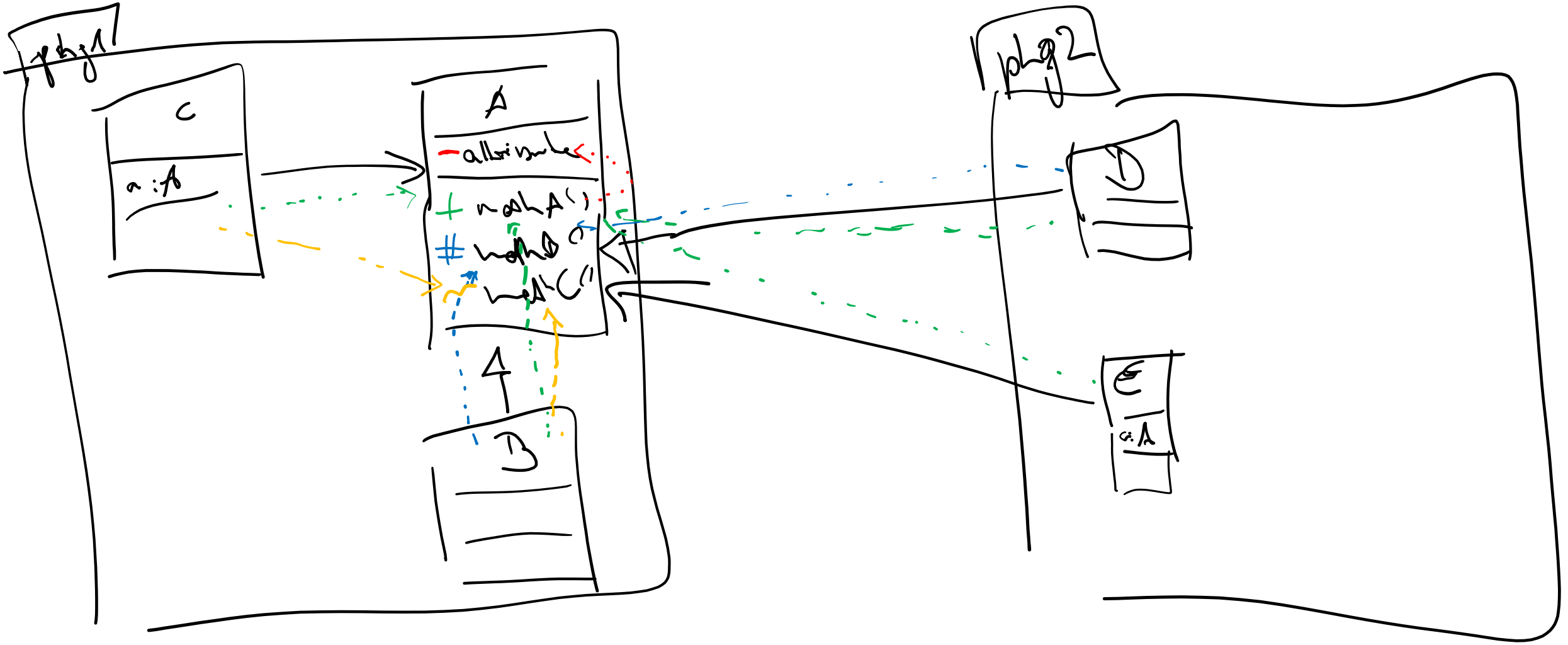
Visibility modifier

- To facilitate encapsulation, the visibility of classes, attributes and methods can be restricted with visibility modifiers.
- Attributes almost always have private visibility.
  - Instead of being directly manipulated by others, they should only be changed by methods in the same class.
  - Ensures that you have control over the attributes' values and can prevent any invalid values that may lead to errors.
- Methods may have any of the visibilities, depending on who they are providing functionality to:

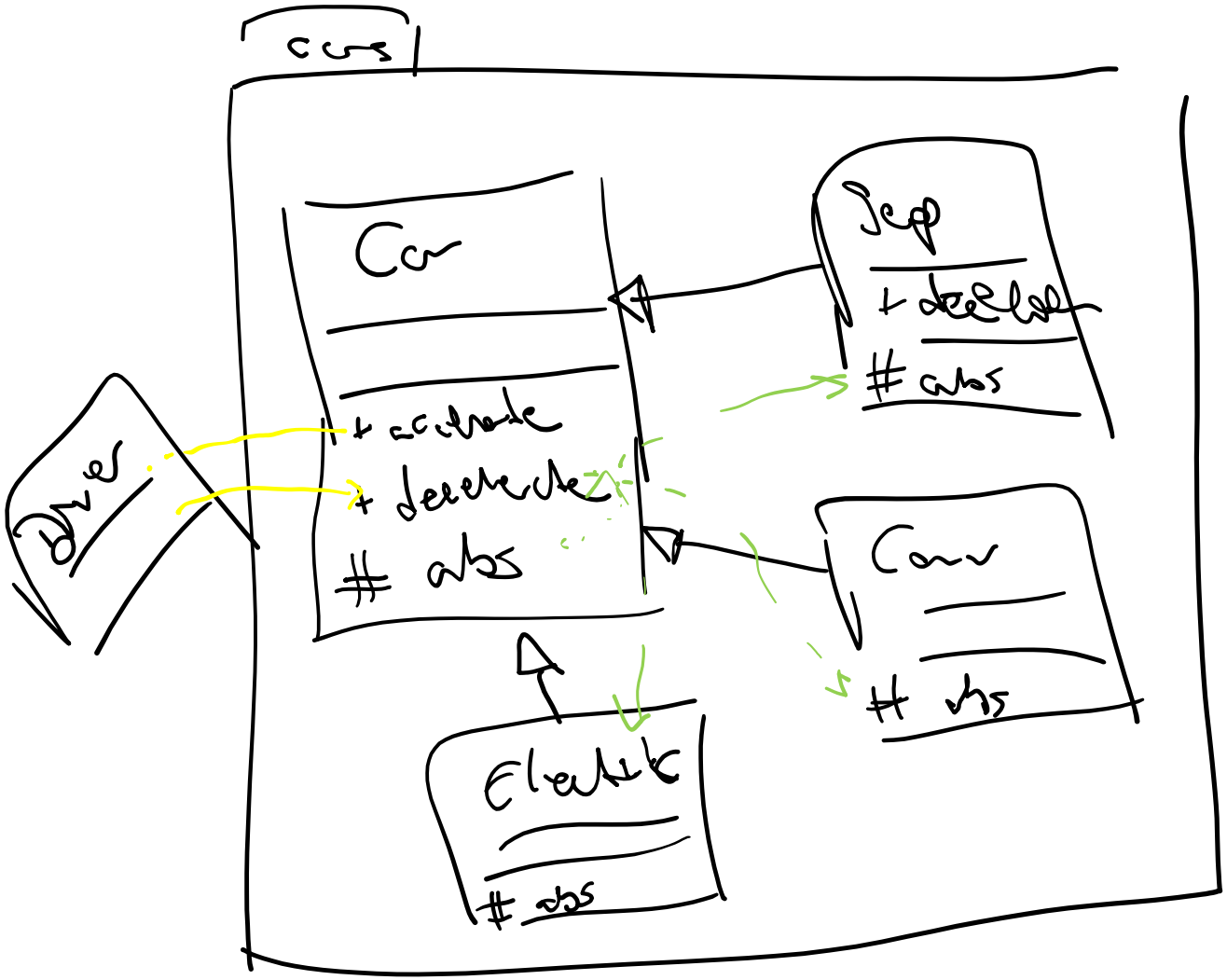


Miles, Hamilton: Learning UML 2.0. O'Reilly, 2006

Recap: Visibility Levels



Recap: Protected Visibility Example



In-Class Quiz #9 Solution: OO Design Principles

UNIVERSITY OF ICELAND

a) Single Responsibility Principle

2. An object's implementation should focus on carrying out that object's single responsibility.

b) Separation of Concerns

4. Classes should depend on each other as little as possible.

c) Don't Repeat Yourself

1. Abstract out things that are common and place them in a single location.

d) Program to Interfaces

5. Don't rely on details of an implementation, but only on what is defined by an interface.

e) Open/Closed Principle

3. Classes should be open for extension, but closed for modification.

f) Liskov Substitution Principle

6. Superclasses should be substitutable by subclasses without changing expected behavior.

Software Testing Basics

see also:

Head First Software Development, Chapter 7

What is Testing?

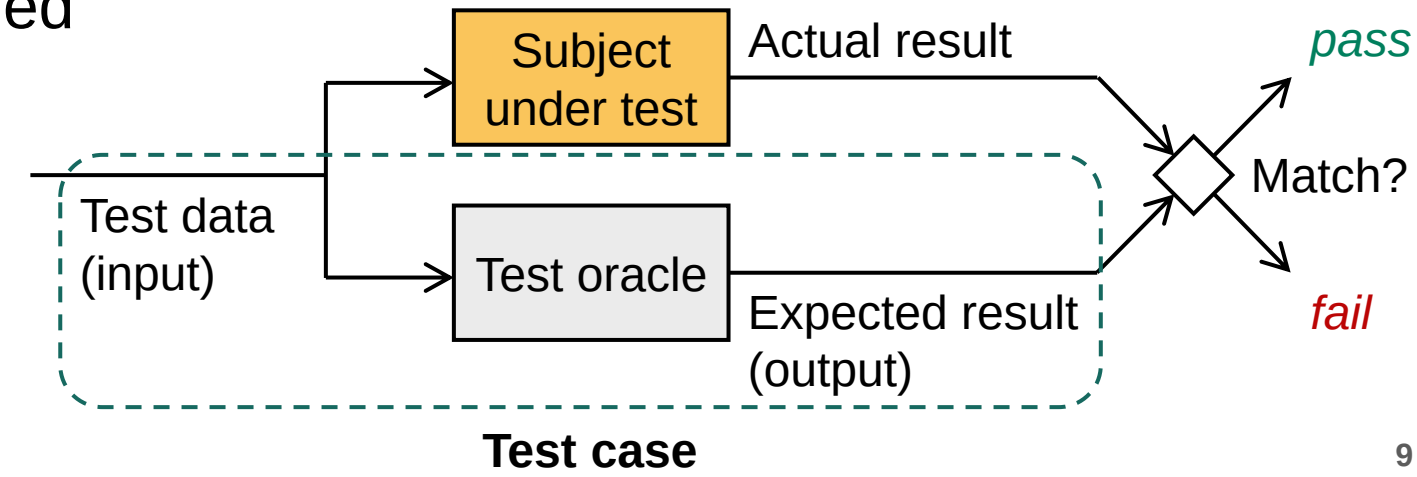
- A software exhibits **quality** if it satisfies the customer's requirements.
  - "Does the right thing, and does it right."
- A **defect** is a deviation from expected (specified) behavior.

- **Testing** means checking whether a software contains defects
  - by analyzing the source code (static testing)
  - by executing the software (dynamic testing)
- **Debugging** means identifying the reasons for defects and fixing them.
  - Success is confirmed by re-testing.

- Testing is only one part of a larger set of quality management activities.
  - Mechanisms for prevention and correction of mistakes should permeate whole SW process.

Basic Testing Concepts

- A **test case** defines the expected result (output) that the system is expected to produce if exposed to particular test data (input).
  - More generically, we can also speak of **pre- and post-conditions** of a test.
- The expected result of a test case (i.e. the reference for correct system behavior) is provided by a **test oracle**.
  - Typically, a human who is carefully interpreting the requirements / specification / source code
    - Caution: Tests can be defective too! (Are we testing the right thing? Are we testing it right?)
- When running the test, the **subject under test** is exposed to the test data, and the actual result is compared to the expected result.
  - The test is said to **pass** if both results match,
  - and is said to **fail** otherwise.



Functional vs. Structural Testing

UNIVERSITY OF ICELAND

Functional ("Black-Box") Testing

- Test cases based only on requirements / specification
- Test quality depends on precision of specification
- Specified, but not implemented behavior is discovered
- Implemented, but not specified behavior is not tested

Structural ("White-Box") Testing

- Test cases based on structural analysis of system's source code
- Test quality depends on coverage of large number of execution paths
- Implemented behavior can be tested without specification
- Mismatches between specification and implementation not discovered

Testing Philosophies: Code First

Approach

For each component:

- Code component
- Code test cases
- *Repeat…*
  - Run tests
  - Debug component
- *…until tests pass*

Issues

- We can get carried away while coding
  - Build too much at once
  - Build things that aren't required
- We find testing boring
  - Tackling the same problem once again
  - Even though we already found a solution
    - …and we are convinced we got it right!
  - Tests perceived as criticism
  - Test results perceived as failure
- We are reluctant to invest effort
  - Testing is done half-heartedly
  - Debugging is done messily

Testing Philosophies: Test First (Test-Driven Development)

Approach

For each component:

- Code test case
- Run test → fails
- Code component
- *Repeat…*
  - Run test
  - Debug component
- *…until test passes*
- Refactor software

Benefits

- We focus on the essentials
  - We start by considering what exactly the requirements mean for our code
  - We code only the minimum solution that will satisfy the test
- Testing is part of the engineering
  - Test results perceived as progress
  - We know we're always on tested ground

- **Issues** that won't go away:
  - Requires discipline and common sense
  - Requires awareness of test complexity

Regression Testing

- A test that passed once will not necessarily pass forever after
- Changes to the tested component (or side effects of changes elsewhere) may break things

- **Regression Testing:** All applicable tests should be run after any change to ensure no defects have been introduced to existing behavior
  - What are “applicable tests” may not be easily decidable
  - Balance time to run tests with coverage of tests

- Can be accomplished with a test automation framework

Break

Test Automation

see also:

Head First Software Development, beginning of Chapter 8

Test Automation

- Executing test cases manually is tedious
  - e.g. feeding parameters into methods and observing resulting states
- Especially when
  - Setting up complex pre-conditions
  - Examining complex post-conditions
  - Repeating test cases for regression testing

➢Recommended to formulate tests as code fragments

  - Can be maintained in conjunction with productive code
  - Can be executed repeatedly by testing framework
- Caution: Test code can be defective too!
  - Keep things simple
  - Know what you are doing

JUnit

- A simple open-source framework for writing and running repeatable tests
  - can be obtained from junit.org
  - also comes with IDEs, e.g. Eclipse
- Write JUnit tests
  - to expose your code to test conditions, and check if resulting conditions match assumptions
  - Rec. practice: Put test classes in a **`test`** folder hierarchy mirroring the **`src`** folder hierarchy
    - Unless you want to test package-visible methods – then include tests in **`src`** folder hierarchy
- Compile and run tests from command line…
  - **`javac`** ***`SomeClassTest.java`***
  - **`java org.junit.runner.JUnitCore`** ***`SomeClassTest`***
    - assuming that **`junit.jar`** is on your classpath
- …or easier: from within the IDE
  - e.g. in Eclipse: Run > Run As > JUnit Test

```
src
  com
    xyz
      SomeClass.java
test
  com
    xyz
      SomeClassTest.java
```

A Basic JUnit Test Template

```
import org.junit.*;
import static org.junit.Assert.*;

public class SomeClassTest {

   @Before
   public void setUp() { ... }

   @After
   public void tearDown() { ... }

   @Test
   public void testSomeBehavior() {
      ...
      assert...(...);
   }
}
```

Setup work to be performed before each test (have only one of these)

Cleanup work to be performed after each test (have only one of these)

Test of one particular behavior (have as many of these as necessary)

Assert a certain condition that will indicate whether the test passed (JUnit provides various such methods, e.g. ***assertEquals***, ***assertNotNull***, ***assertSame***, ***assertTrue***...)

A Simple Test-Driven Development Example: **The Test Fixture**

- We want to implement a class that represents a certain amount of money in a certain currency. It should be possible to add, compare and convert amounts.

```
public class MoneyTest {
  private Money isk1000, isk2000;

  @Before
  public void setUp() {
    Money isk1000 = new Money(1000, "ISK");
    Money isk2000 = new Money(2000, "ISK");
  }

  @After
  public void tearDown() {
    isk1000 = null;
    isk2000 = null;
  }
```

Create two objects that our tests will work with

Let garbage collector destroy objects after each test (good practice to prevent side effects between tests)

contd.

A Simple Test-Driven Development Example: **The Test Fixture**

- We want to implement a class that represents a certain amount of money in a certain currency. It should be possible to add, compare and convert amounts.

contd.

```
    @Test
    public void testAmount() {
      assertEquals(1000, isk1000.getAmount());
    }

    @Test
    public void testCurrency() {
      assertEquals("ISK", isk1000.getCurrency());
    }
}
```

Test if parameters passed to constructor are stored by object and retrievable through getters

assertEquals fails if the first (expected result) parameter does not match the second (actual result) parameter

Initially, we can't even compile this test since it refers to classes and methods that do not yet exist.

A Simple Test-Driven Development Example: **Creating a Skeleton of the Class under Test**

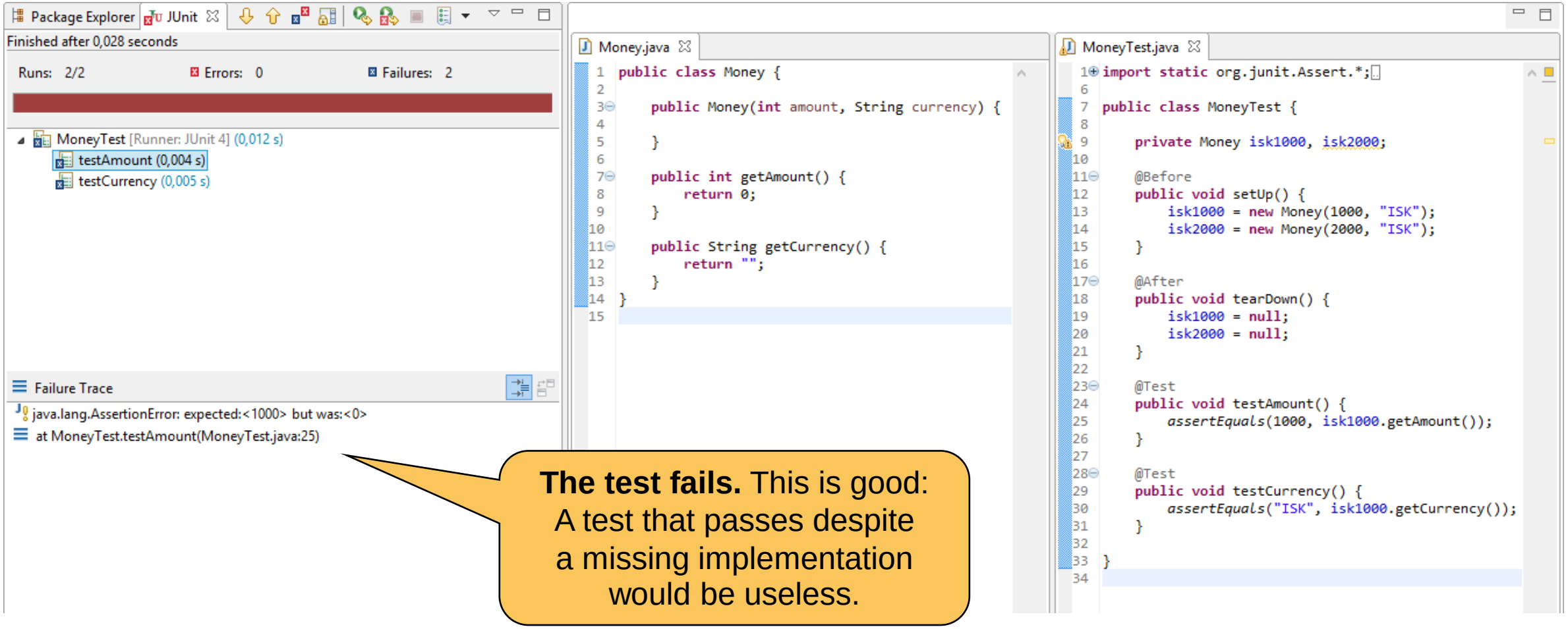
- We want to implement a class that represents a certain amount of money in a certain currency. It should be possible to add, compare and convert amounts.

```
public class Money {

  public Money(int amount, String currency) {
  }

  public int getAmount() {
    return 0;
  }

  public String getCurrency() {
    return "";
  }

}
```

A skeletal implementation that obviously does not do anything yet, but at least lets the test case compile.

A Simple Test-Driven Development Example: **Running the Test for the First Time**

- Run **`MoneyTest`** as JUnit test



The test fails. This is good: A test that passes despite a missing implementation would be useless.

A Simple Test-Driven Development Example: **Implementing the Class under Test**

- We want to implement a class that represents a certain amount of money in a certain currency. It should be possible to add, compare and convert amounts.

```
public class Money {

  private int amount;
  private String currency;

  public Money(int amount, String currency) {
    this.amount = amount;
    this.currency = currency;
  }
```

A simple implementation of the behavior expected by the test.

contd.

A Simple Test-Driven Development Example: **Implementing the Class under Test**

- We want to implement a class that represents a certain amount of money in a certain currency. It should be possible to add, compare and convert amounts.

contd.

```
    public int getAmount() {
        return amount;
    }

    public String getCurrency() {
        return currency;
    }

}
```

- **Note: No setters!** We didn't need them to satisfy the test, so we didn't implement them.
- **Key principle:** Implement only what is necessary to pass the test!
- If you feel there should be more functionality,
  - write a test for it first
  - then write the code to satisfy that test

A Simple Test-Driven Development Example: **Running the Test Again**

- Run **`MoneyTest`** as JUnit test

Test Execution Order

- A possible execution order:
  - `setUp()`
  - `testAmount()`
  - `tearDown()`
  - `setUp()`
  - `testCurrency()`
  - `tearDown()`
- Another possible execution order:
  - `setUp()`
  - `testCurrency()`
  - `tearDown()`
  - `setUp()`
  - `testAmount()`
  - `tearDown()`

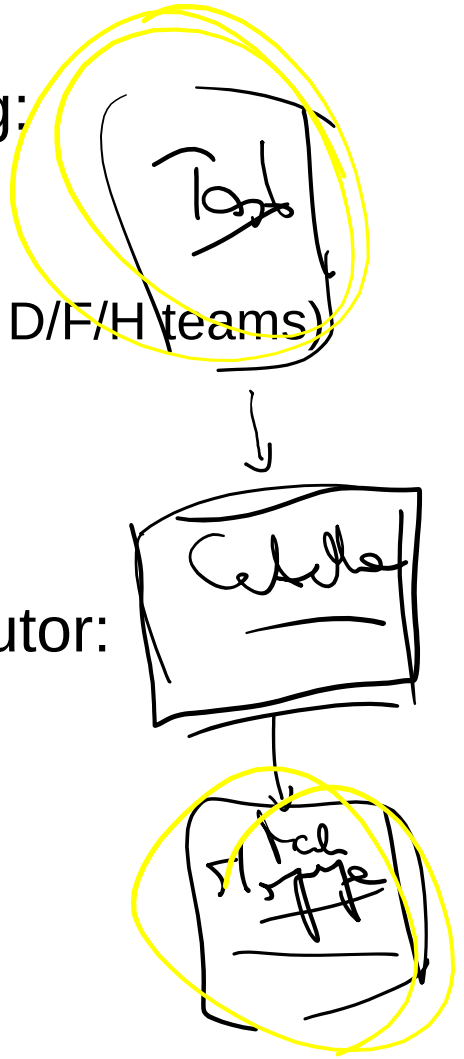
- The execution order of the various `@Test` methods is undefined.
- Don’t assume a particular execution order!
- In particular, don’t rely on the state of the system after execution of one test as the pre-condition for another test!
- Use `@Before` and `@After` methods to set up / clean up system state, and ensure same conditions for each test.

Some Dos and Don’ts

- Do the test classes need to mirror the production classes exactly?
  - No, it can make sense to have a test class that tests the combined behavior of several production classes.
- Can I check several conditions (i.e. make several assertions) in the same test (method)?
  - Yes, but it is not recommended (and JUnit will only report the first violation).
  - You should have one test method per condition you want to test.
- Is there a place to put initialization/teardown code that should only be executed once per test class run (i.e. not before/after each individual test case)?
  - Yes, you can place it in methods with `@BeforeClass` and `@AfterClass` annotations.
    - Order: `@BeforeClass`, `@Before`, `@Test`, `@After`,…, `@Before`, `@Test`, `@After`, `@AfterClass`
    - But think twice if you really need it – usually means your tests are too tightly coupled

Team Assignment #4: Software Tests

- By **Sun 27 Mar**, submit in Canvas a **test document (PDF)** showing:
  - **A test fixture** you have implemented for one of your controller components
    - Provide source code of test fixture based on Junit
  - **A mock object** you have implemented to simulate a storage component (for D/F/H teams) or another team's component (for T teams)
    - Provide documented source code of mock object



- On **Wed 30 Mar**, present and **explain** your test document to your tutor:
  - Why did you choose the test cases (inputs and expected results) you did?
  - Why do you believe the behavior covered by your test cases is sufficient?
  - How does your mock object simulate the mocked object's behavior?

Team Assignment #4: Software Tests

- **Grading criteria** (33.33% each)
  - The **test fixture** provides the proper **structural framework** for testing a controller's method
    - It ensures a clean environment for each test using @Before and @After methods and passes suitable mock object instances to the controller under test.
    - The controller under test would not need to be changed between testing and productive use.
  - The **test fixture** exposes a controller's method to plausible **test cases**
    - The tests feed suitable input to the method for asserting properties of its output that express the method's expected behavior in different scenarios (e.g. successful and unsuccessful search).
    - The tests do not test behavior of the mock objects.
  - **Mock objects** are used in a plausible way.
    - The mock objects simulate (rather than realize) complex behavior that the controller under test relies on.
    - Several mock objects are used to simulate different behaviors.

Team Assignment #4: Software Tests

- **Deadlines (mandatory for assignment grade / optional for bonus grade)**
  - by **Wed 23 Mar 12:00**, submit your anonymized draft to the Peer Feedback Assignment #4
    - the earlier you submit, the more time your reviewers have to give you feedback
  - on **Wed 23 Mar 15:00-18:15**, discuss your draft with your tutor
  - by **Fri 25 Mar 23:59**, submit your peer feedback on the drafts assigned to you
    - the earlier you submit, the more time the other students have to incorporate your feedback
  - by **Sun 27 Mar 23:59**, submit your final PDF document to the Team Assignment #4
  - by **Wed 30 Mar 12:00**, rate the quality / usefulness of the peer feedback you received
  - on **Wed 30 Mar 15:00-18:15**, present and explain your test cases to your tutor

In-Class Quiz #10: Functional vs. Structural Testing

- **Indicate whether the following statements apply to Functional (Black-Box) Testing or Structural (White-Box) Testing:**

a) Implemented behavior can be tested without specification
b) Implemented, but not specified behavior is not tested
c) Mismatches between specification and implementation not discovered
d) Specified, but not implemented behavior is discovered
e) Test cases based only on requirements / specification
f) Test cases based on structural analysis of system's source code
g) Test quality depends on coverage of large number of execution paths
h) Test quality depends on precision of specification

Thank you!

book@hi.is