UNIVERSITY OF ICELAND

HBV401G SOFTWARE DEVELOPMENT

11. Software Testing

Matthias Book

Spring 2022

FACULTY OF INDUSTRIAL ENGINEERING, MECHANICAL ENGINEERING AND COMPUTER SCIENCE

Recap: What is Testing?

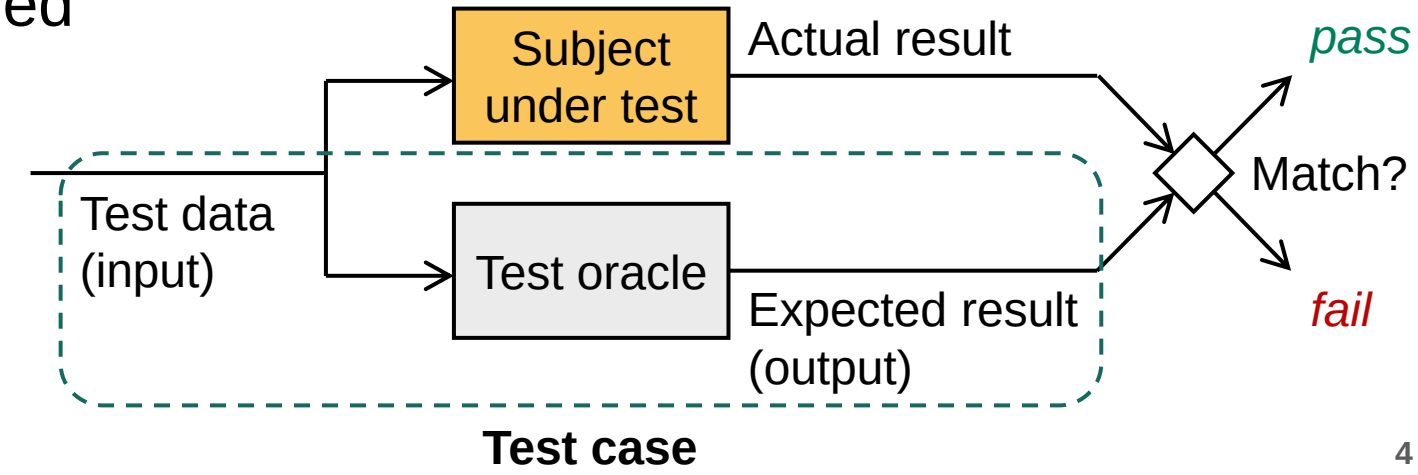
- A software exhibits **quality** if it satisfies the customer's requirements.
  - "Does the right thing, and does it right."
- A **defect** is a deviation from expected (specified) behavior.

- **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.

Recap: 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.



In-Class Quiz #10 Solution: 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	Structural
b)	Implemented, but not specified behavior is not tested	Functional
c)	Mismatches between specification and implementation not discovered	Structural
d)	Specified, but not implemented behavior is discovered	Functional
e)	Test cases based only on requirements / specification	Functional
f)	Test cases based on structural analysis of system's source code	Structural
g)	Test quality depends on coverage of large number of execution paths	Structural
h)	Test quality depends on precision of specification	Functional

Recap: 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

Recap: A Basic JUnit Test Fixture

```
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***...)

UNIVERSITY OF ICELAND

Mock Objects

see also:

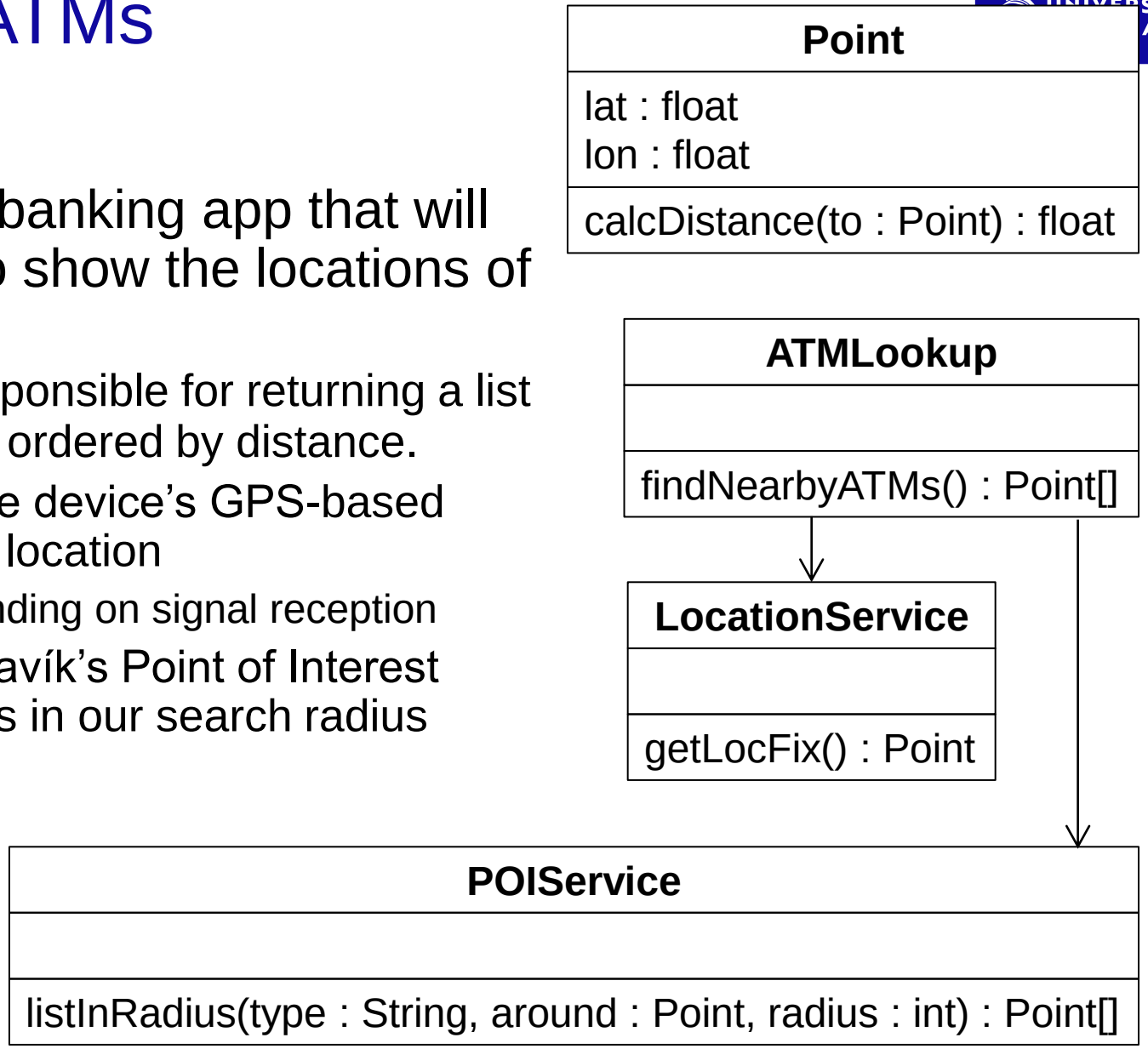
Head First Software Development, end of Chapter 8

Mock Objects

- While testing a class, not all classes that this class depends on might be implemented yet.
  - e.g. accessing a database, an external API, a mobile device sensor…
- We can replace the missing implementation with a **mock object** *that looks like the expected object, but provides only simulated behavior.*
- The mock object might need to simulate several types of behaviors to test different scenarios
  - e.g. requested data found, requested data not found, data source not accessible etc.
  - Rather than implementing all behaviors in one object, we use several similar mock objects
    - Mock frameworks can help us to produce these.

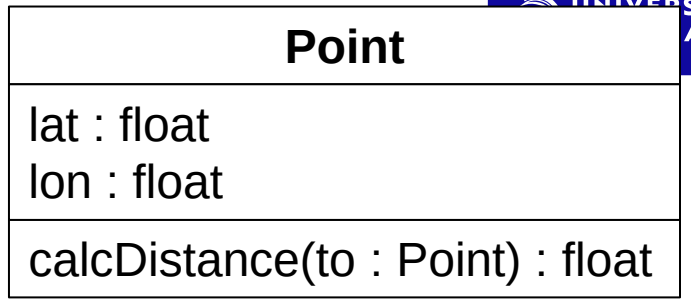
Example: Finding Nearby ATMs
Requirements

- Assume we are building a mobile banking app that will (among other features) allow us to show the locations of nearby ATMs on a map.
  - The **`ATMLookup`** class should be responsible for returning a list of ATM coordinates in a 5 km radius, ordered by distance.
  - To do this, it should rely on the mobile device's GPS-based location service to obtain our current location
    - which it may or may not work, depending on signal reception
  - And it should query the City of Reykjavík's Point of Interest (POI) service for all locations of ATMs in our search radius
    - which returns POIs unordered
  - All classes use the **`Point`** class to express geographic coordinates
    - which also provides a geographic distance calculator

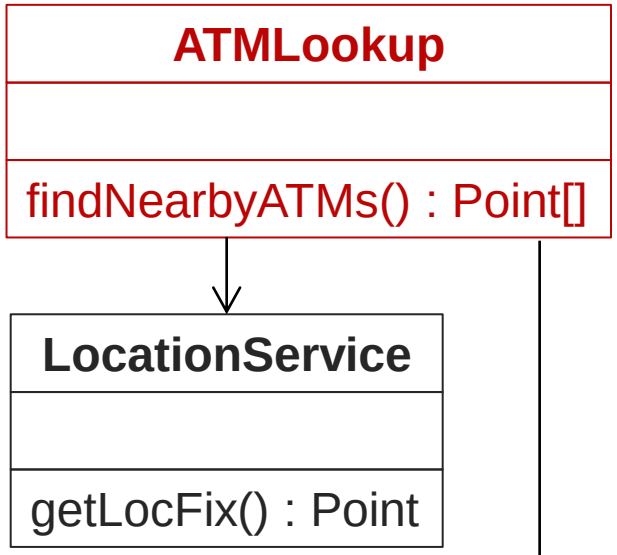


Example: Finding Nearby ATMs
Our Situation

- Assume the **`Point`** class is already present and tested.



- We currently want to implement and test the **`ATMLookup`** class.



- But the **`LocationService`** is currently unavailable since we're not working on a mobile device…
- …and the **`POIService`** has not been implemented yet.

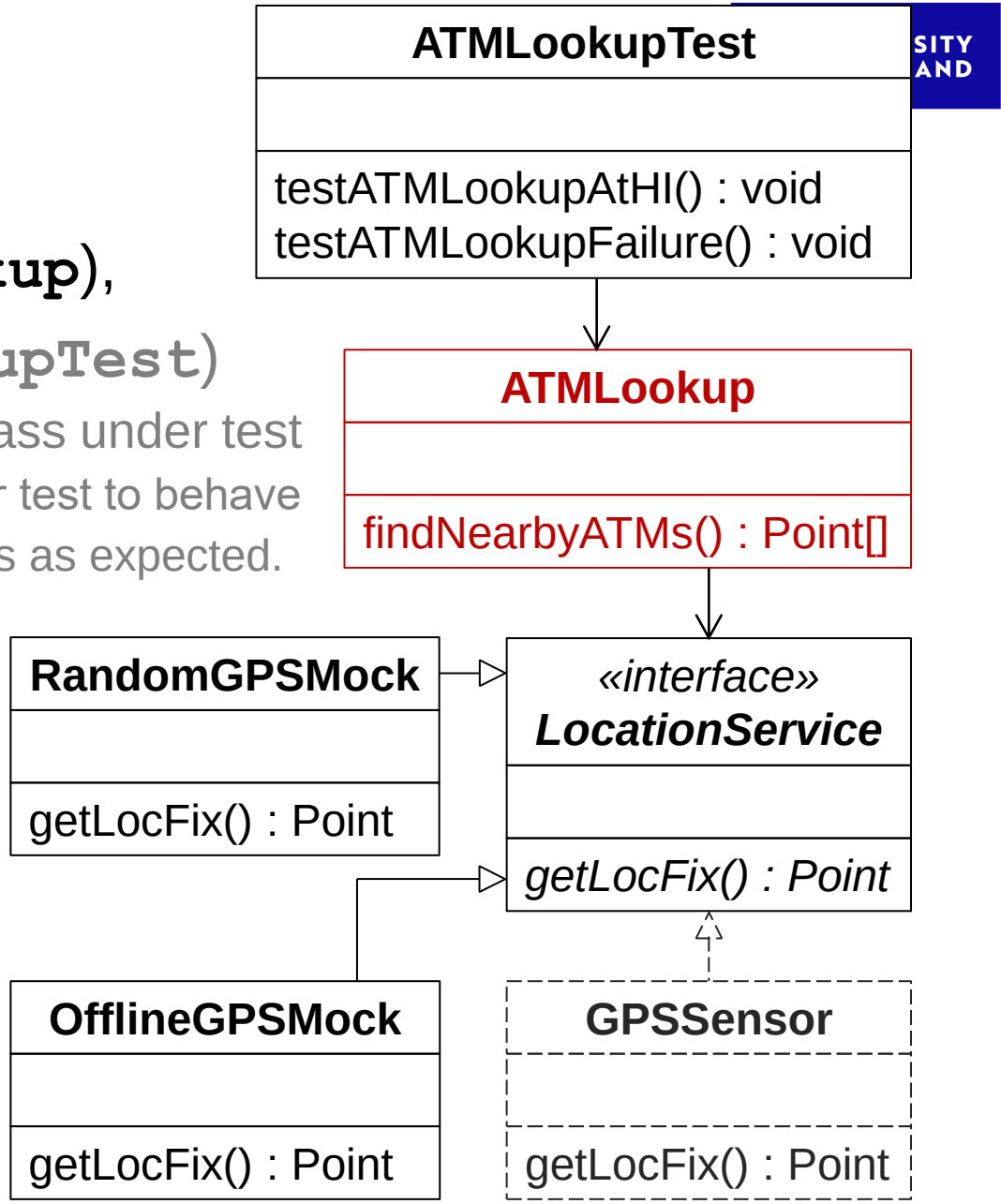
Example: Finding Nearby ATMs
Test Setup (Part 1)

To test a class (the "**class under test**": **`ATMLookup`**),

- we need a **test driver / test fixture** (**`ATMLookupTest`**)
  - that will simulate the classes that will later call the class under test
    - i.e. that specifies how we're expecting the class under test to behave
    - and that checks if the class under test indeed behaves as expected.

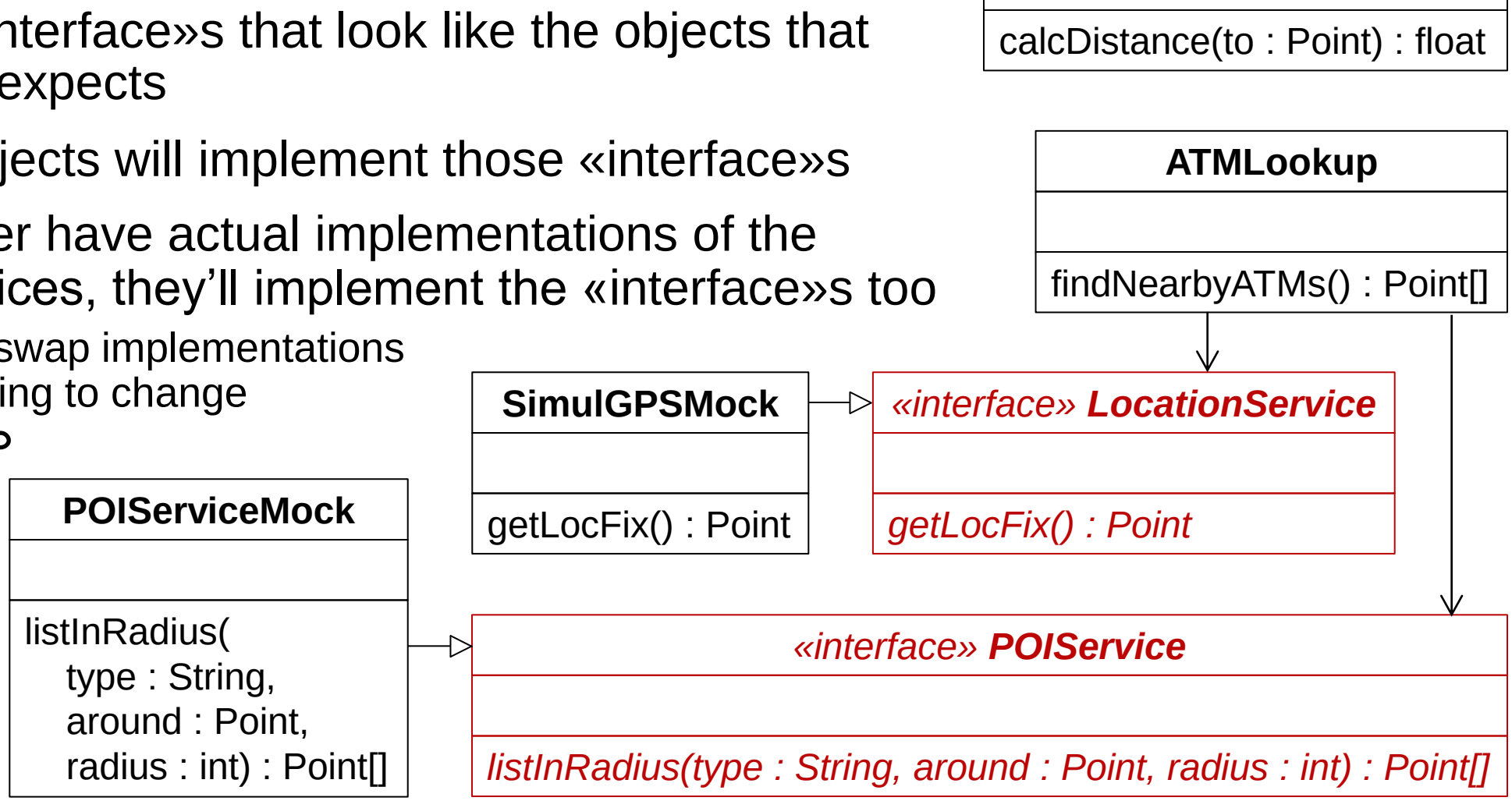
❖we need **mock objects** (**`RandomGPSMock`**, **`OfflineGPSMock`** etc.)

- that will simulate the classes that will be called by the class under test
  - i.e. that satisfy the same «interface» (**`LocationService`**) as the class that will provide the actual behavior later (**`GPSSensor`**)
  - and that can be swapped out to test different scenarios without having to change the class under test.



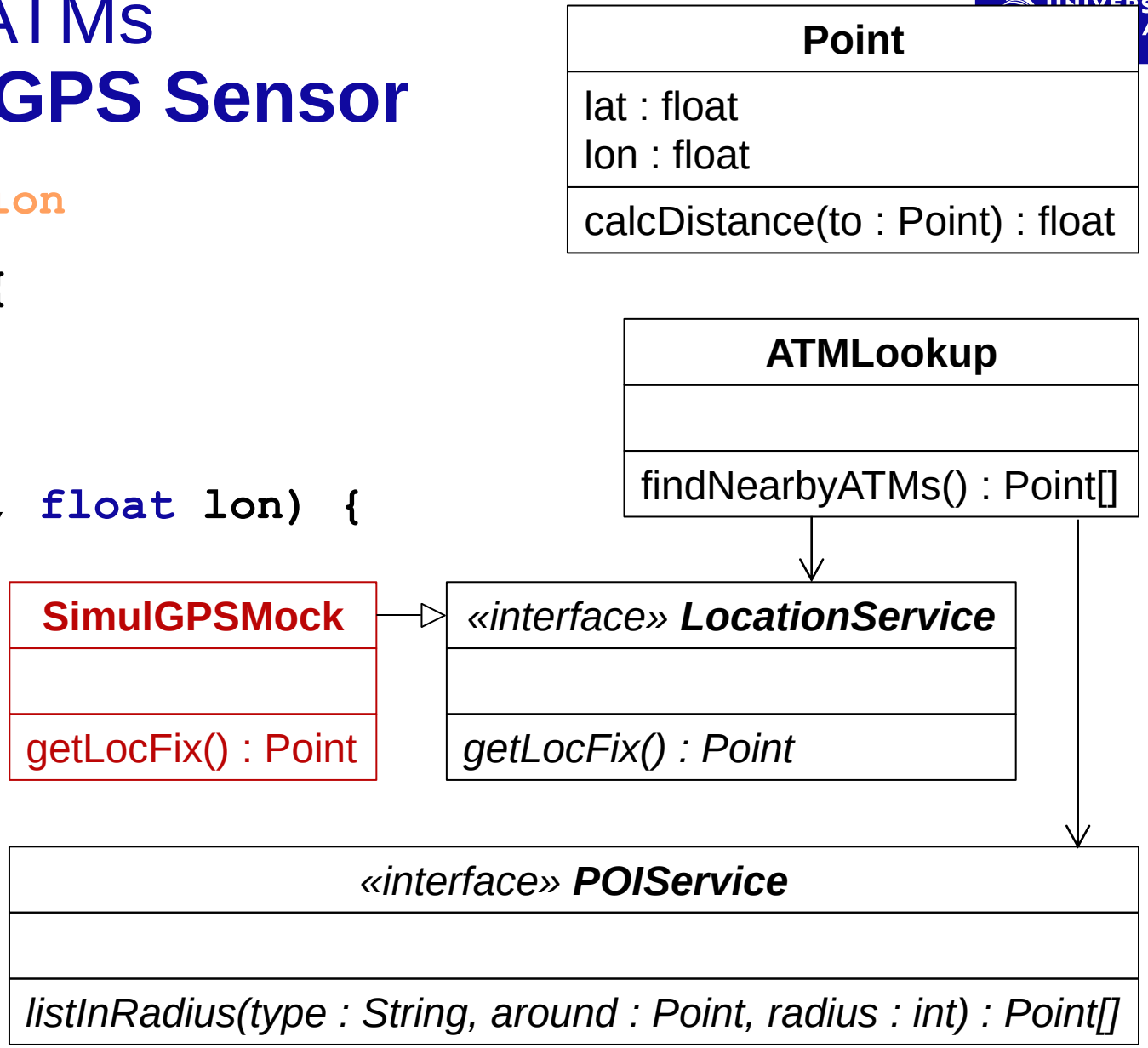
Example: Finding Nearby ATMs
«Interface»s for Mock (and Real) Objects

- We create «interface»s that look like the objects that **`ATMLookup`** expects
- Our mock objects will implement those «interface»s
- When we later have actual implementations of the mocked services, they'll implement the «interface»s too
  - So we can swap implementations without having to change **`ATMLookup`**



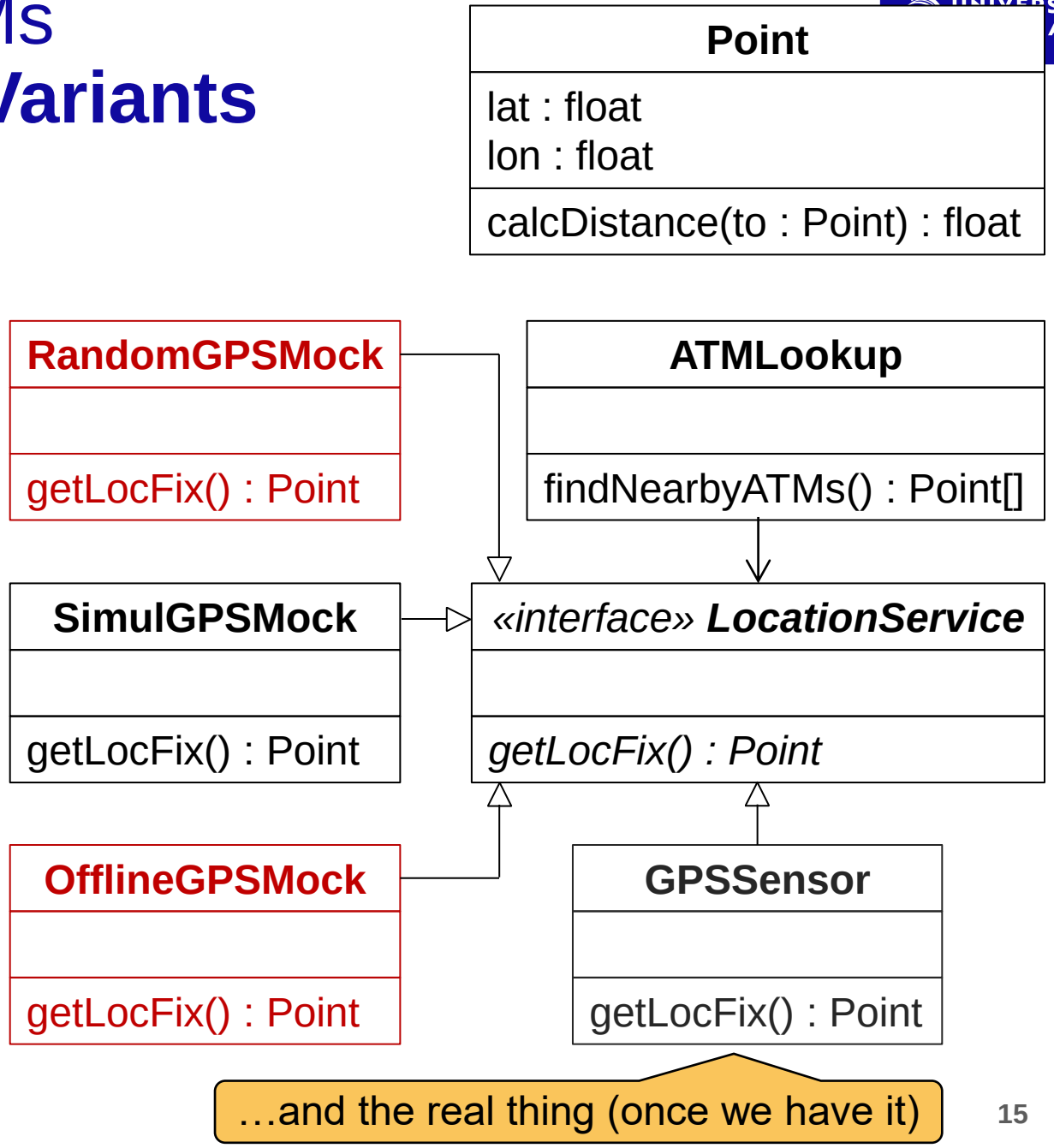
Example: Finding Nearby ATMs
Mock Implementation of GPS Sensor

```
// simulates fix at given location
public class SimulGPSMock
    implements LocationService {

  private float lat;
  private float lon;

  public SimulGPSMock(float lat, float lon) {
    this.lat = lat;
    this.lon = lon;
  }

  public Point getLocFix()
      throws OfflineException {
    return new Point(lat, lon);
  }

}
```



Example: Finding Nearby ATMs
Mock GPS Implementation Variants

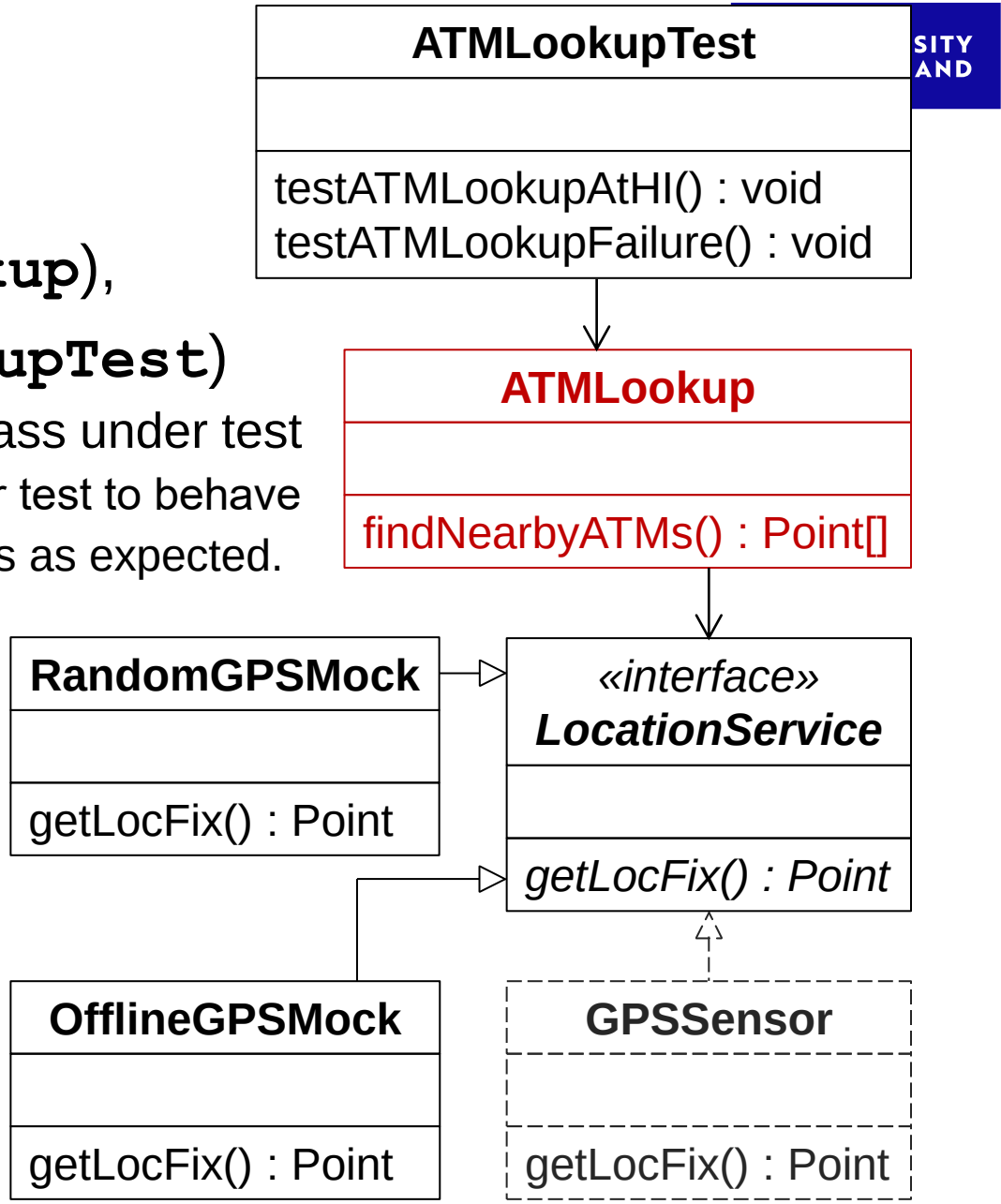
```
// simulates random fix
public class RandomGPSMock
     implements LocationService {
   public Point getLocFix()
        throws OfflineException {
     return new Point(
       Math.random()*180-90,
       Math.random()*360-180);
   }
}

// simulates unavailable sensor
public class OfflineGPSMock
     implements LocationService {
   public Point getLocFix()
        throws OfflineException {
     throw new OfflineException();
   }
}
```



Example: Finding Nearby ATMs
Test Setup (Part 2)

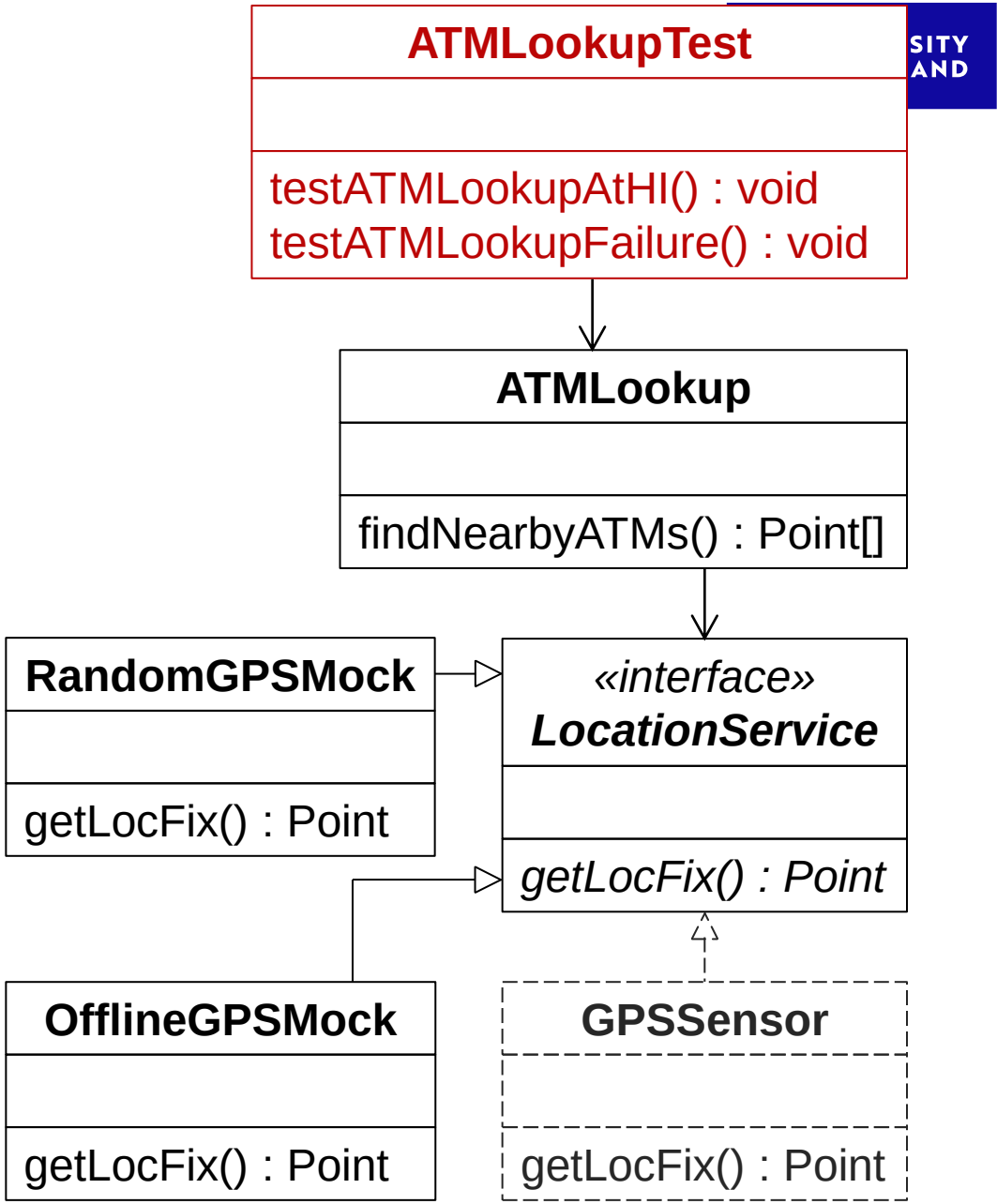
To test a class (the "**class under test**": **`ATMLookup`**),

- ❖ we need a **test driver / test fixture** (**`ATMLookupTest`**)
  - that will simulate the classes that will later call the class under test
    - i.e. that specifies how we're expecting the class under test to behave
    - and that checks if the class under test indeed behaves as expected.
- ✓ we need **mock objects** (**`RandomGPSMock`**, **`OfflineGPSMock`** etc.)
  - that will simulate the classes that will be called by the class under test
    - i.e. that satisfy the same interface (**`LocationService`**) as the class that will provide the actual behavior later (**`GPSSensor`**)
    - and that can be swapped out to test different scenarios without having to change the class under test.



Example: Finding Nearby ATMs
Test Fixture

```
public class ATMLookupTest {
   @Before
   public void setUp() { ... }

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

   @Test
   public void testATMLookupAtHI() {
      ...
   }

   @Test
   public void testATMLookupFailure() {
      ...
   }
}
```



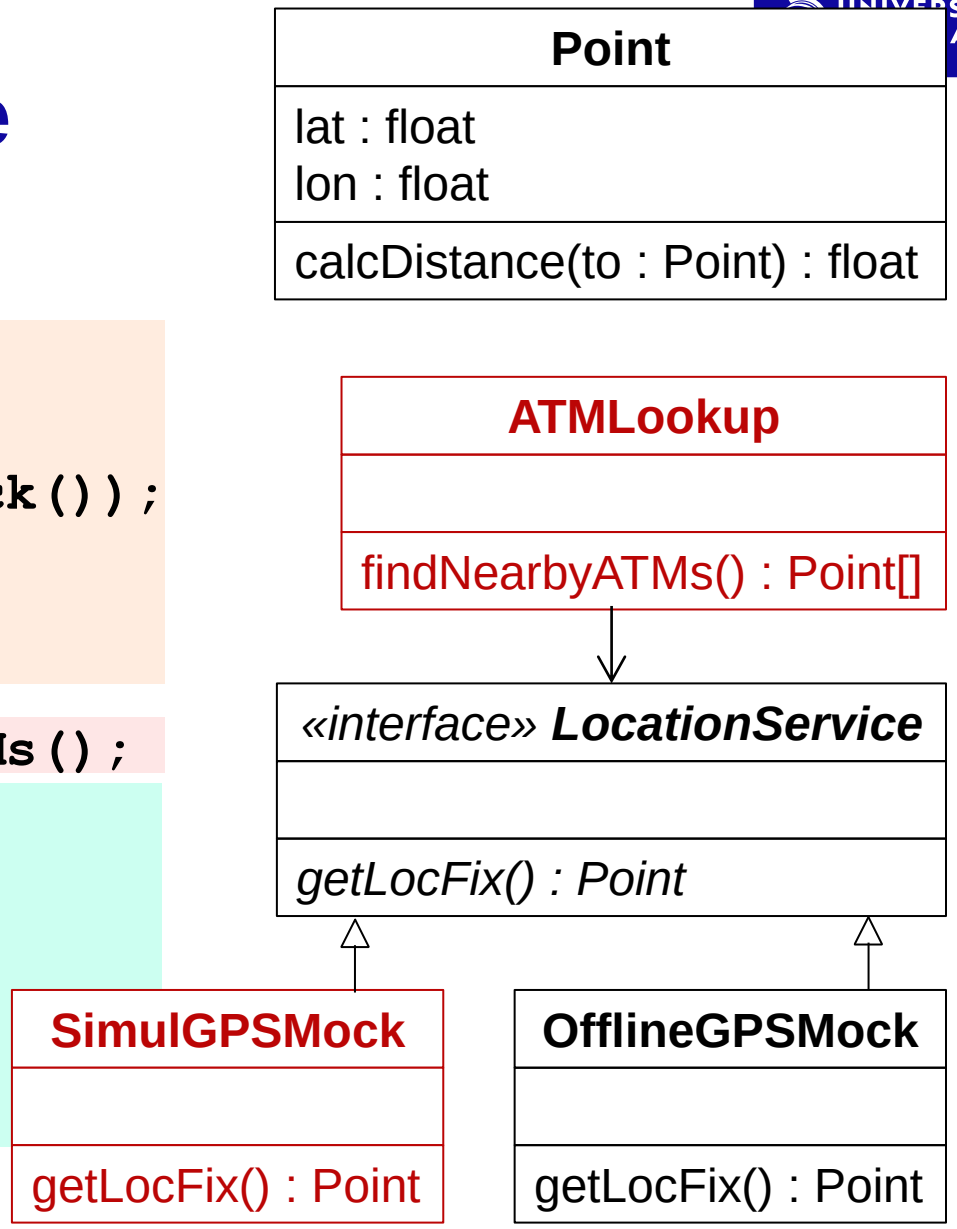
Example: Finding Nearby ATMs
Using Mock Objects in Test Fixture

Test setup

Test call

Check assertions

```
@Test
public void testATMLookupAtHI() {
  LocationService gpsMock =
    new SimulGPSMock(64.139903, -21.950340);
  ATMLookup atmLookup =
    new ATMLookup(gpsMock, new POIServiceMock());
  Point myLoc = gpsMock.getLocFix();
  float prevDist = 0;
  float currDist = 0;

  List<Point> atms = atmLookup.findNearbyATMs();
  assertNotNull(atms);
  for (Point point : atms) {
    currDist = myLoc.calcDistance(point);
    assertTrue(currDist <= 5000);
    assertTrue(currDist >= prevDist);
    prevDist = currDist;
  }
}
```



Example: Finding Nearby ATMs
Using Mock Objects in Test Fixture

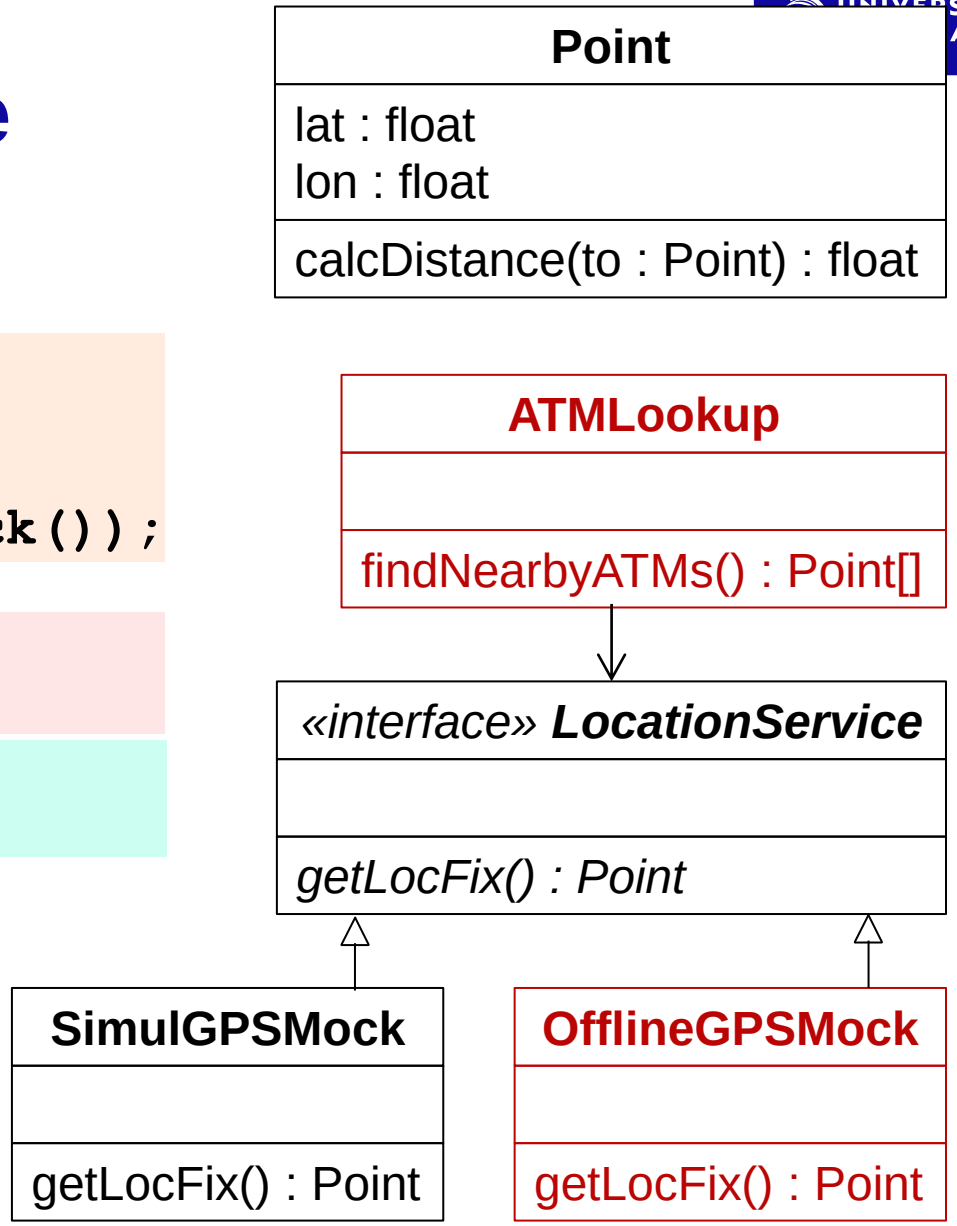
Test setup

Test call

Check assertions

```
@Test
public void testATMLookupFailure() {
  LocationService gpsMock =
    new OfflineGPSMock();
  ATMLookup atmLookup =
    new ATMLookup(gpsMock, new POIServiceMock());

  List<Point> atms =
    atmLookup.findNearbyATMs();
  assertNotNull(atms);
  assertEquals(0, atms.size());
}
```

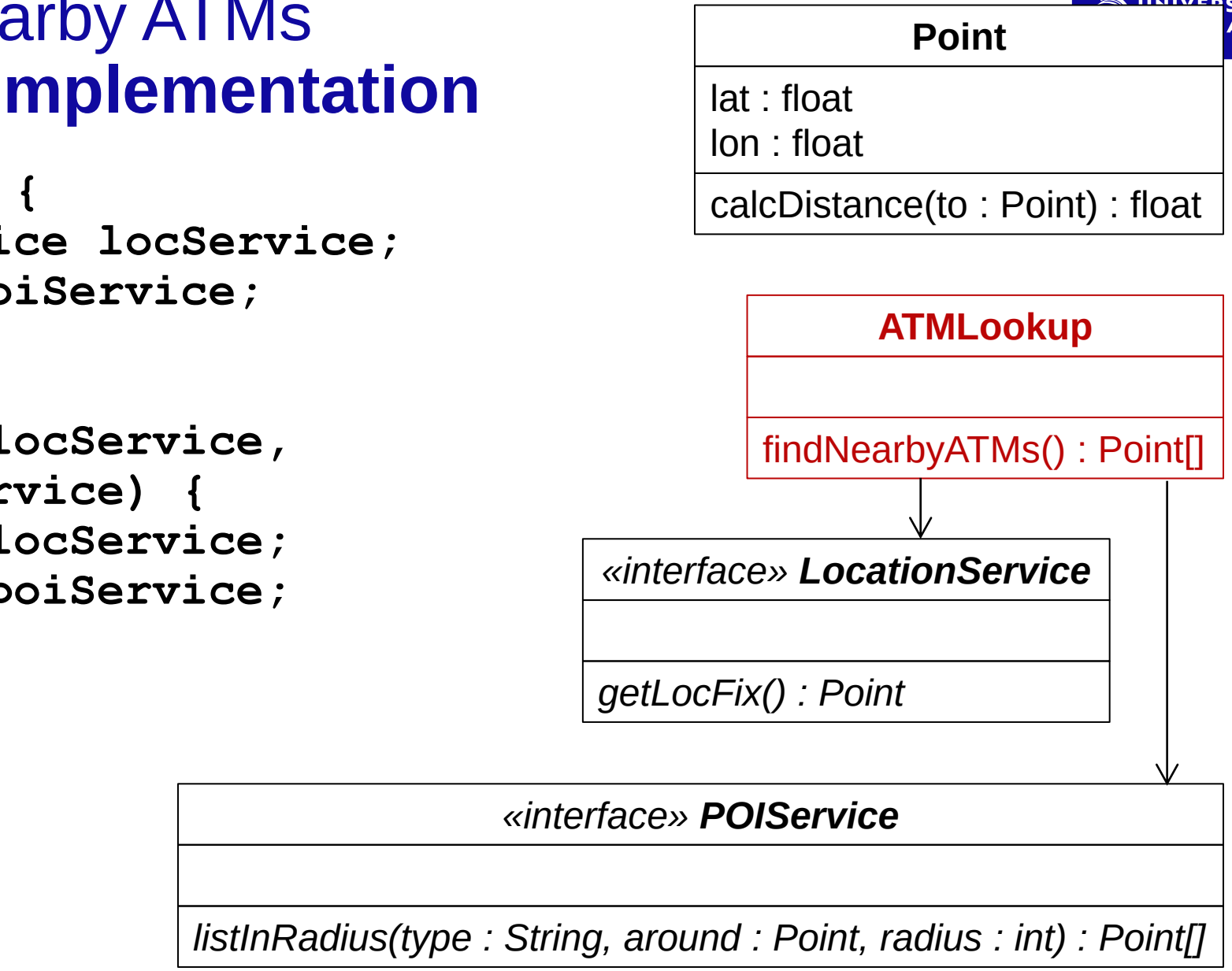
The test defines how we would like to deal with an unavailable GPS signal: Rather than showing an error message to the user, we just won't show any ATMs on the map.



Example: Finding Nearby ATMs
Mock-Independent Implementation

```
public class ATMLookup {
  private LocationService locService;
  private POIService poiService;

  public ATMLookup(
      LocationService locService,
      POIService poiService) {
    this.locService = locService;
    this.poiService = poiService;
  }
```

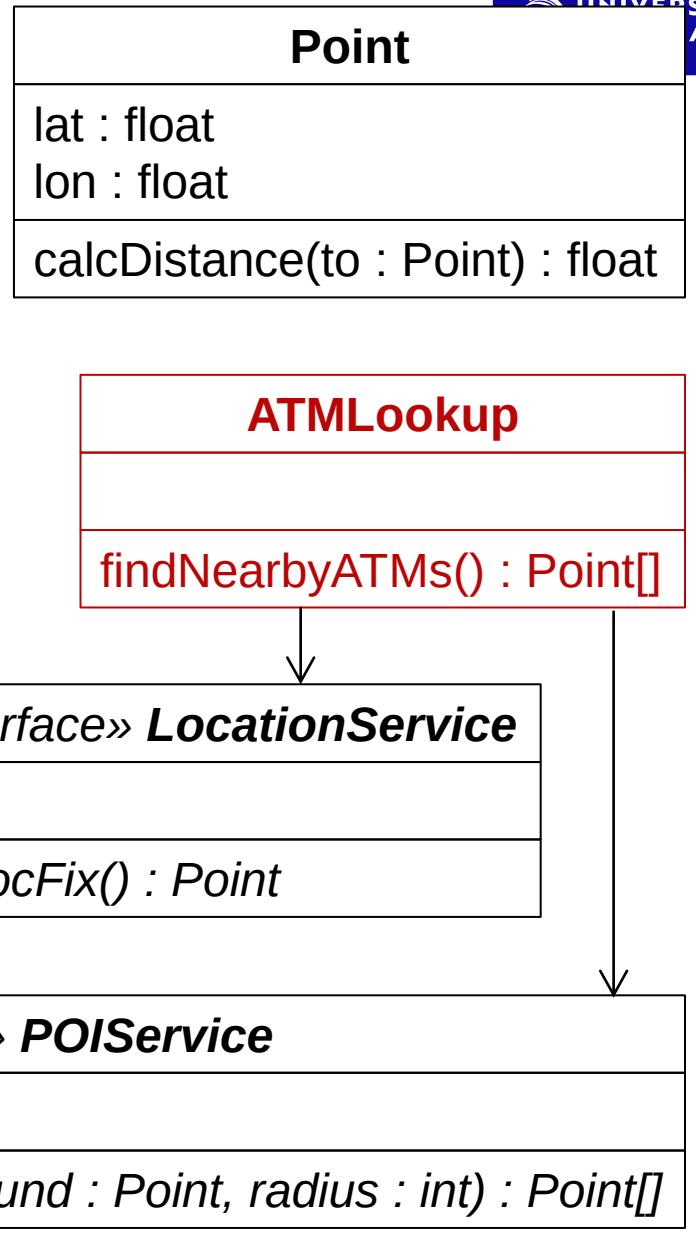
contd.



Example: Finding Nearby ATMs
Mock-Independent Implementation

contd.

```
public List<Point> findNearbyATMs() {
  List<Point> atms = new ArrayList<Point>();
  try {
    Point myLoc = locService.getLocFix();
    List<Point> pois =
      poiService.listInRadius(
        "ATM", myLoc, 5000);
    // [algorithm to read points from pois,
    // calculate distances to myLoc, and
    // add points to atms in ascending
    // order of distance]
  } catch(OfflineException e) {
  }
  return atms;
}

}
```



In-Class Quiz #11: Software Testing Concepts

- **Indicate which concepts are described by the following definitions – Test Cases, Test Fixtures, Test Oracles, Test Subjects or Mock Objects?**

a) An entity determining what a test subject's expected "correct" output for a particular input is supposed to be.

b) A piece of code exposing a test subject to a particular test case.

c) A piece of code that simulates certain behavior of other, not yet implemented code that a test subject is relying on.

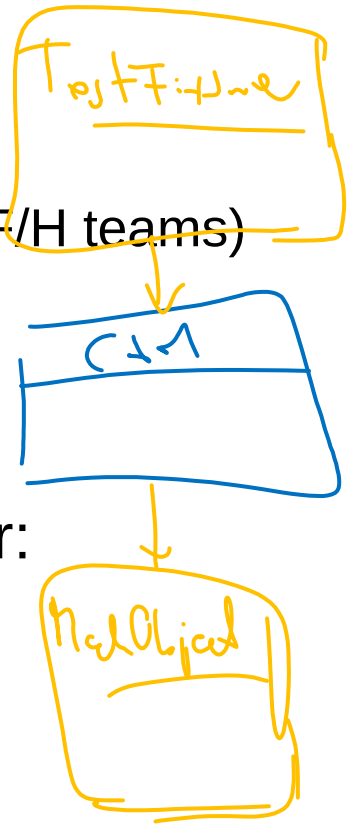
d) A piece of code whose correctness / agreement with specifications is being tested.

e) A specific output/result that is expected to be produced by a test subject when exposed to a specific input.

Break

Recap: 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

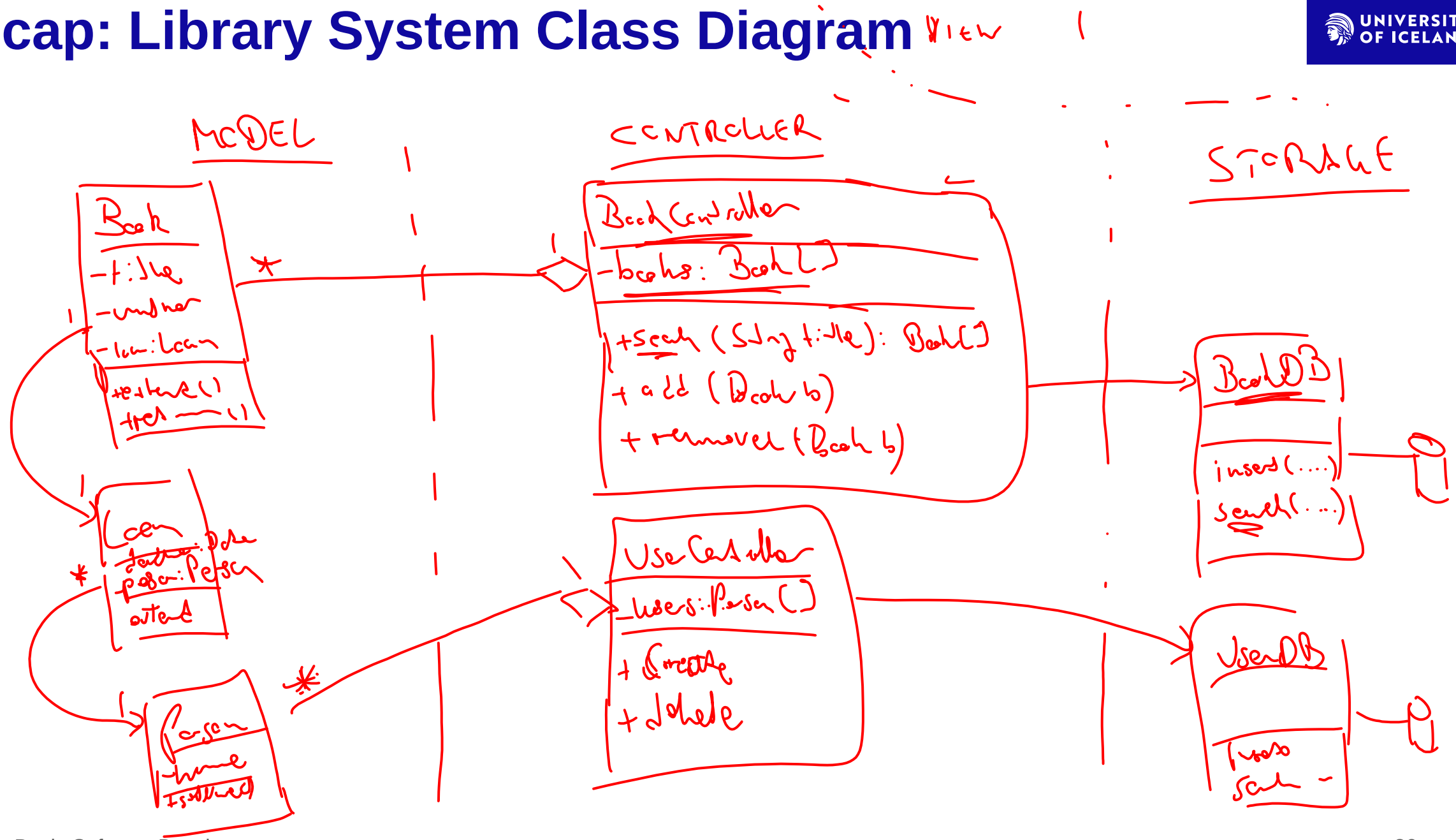


- On **Wed 30 Mar**, present and **explain** your test document to your tutor:
  - Why did you choose the test cases (inputs and expected results) you did?
  - Why do you believe the behavior covered by your test cases is sufficient?
  - How does your mock object simulate the mocked object's behavior?

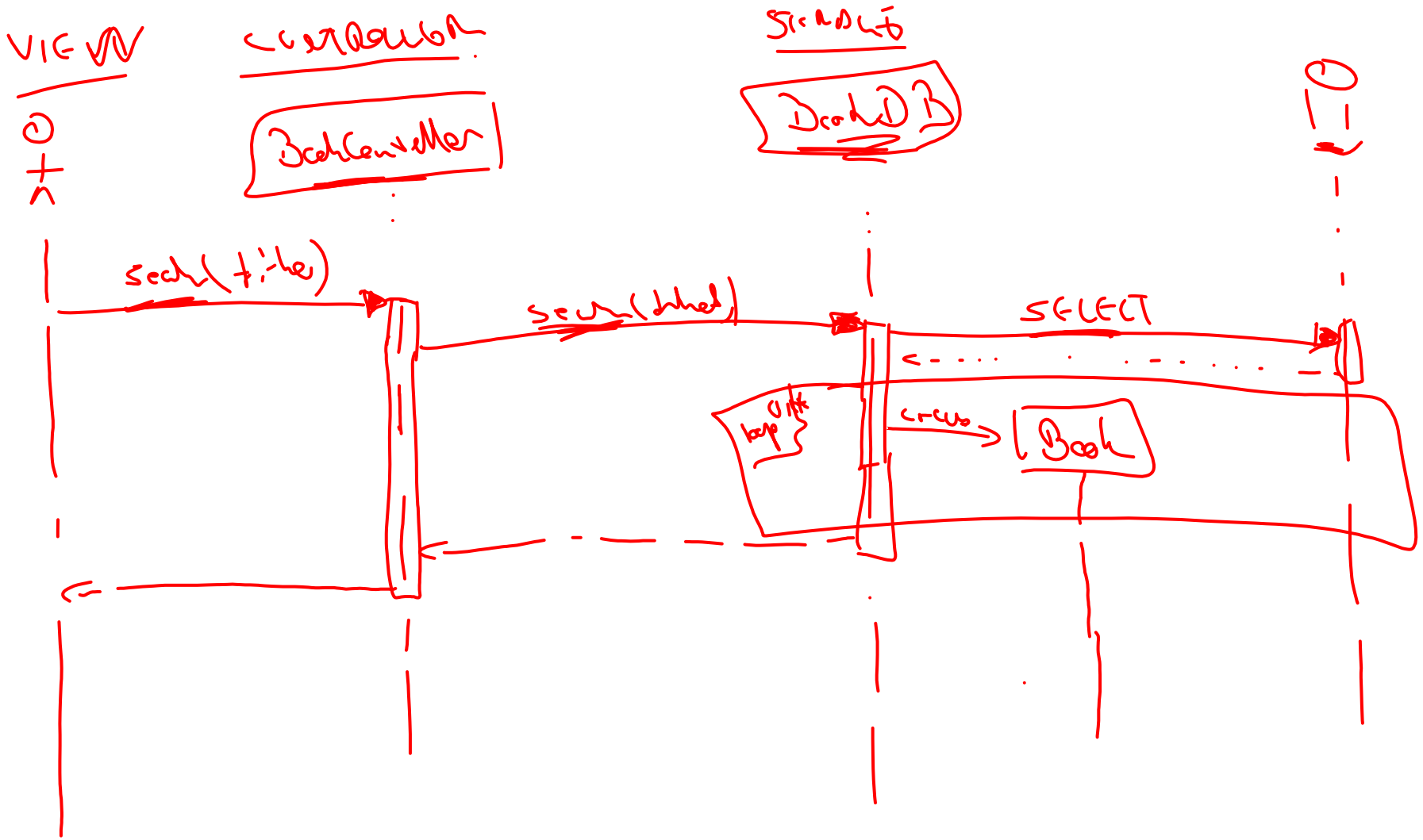
Recap: Team Assignment #4: Software Tests

- **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.

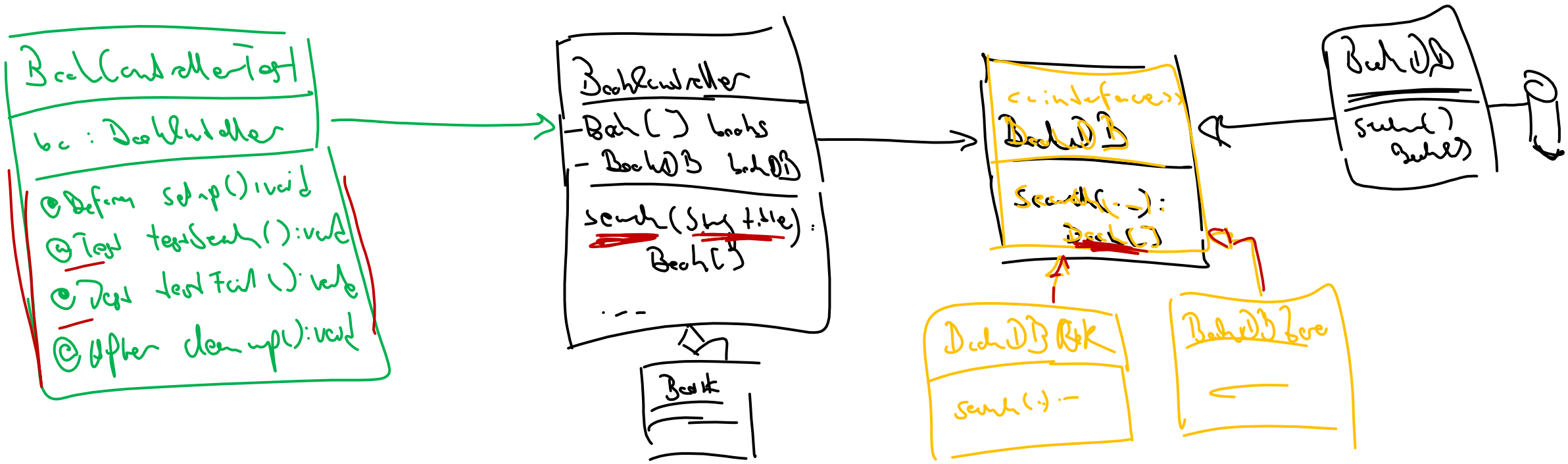
Recap: Library System Class Diagram



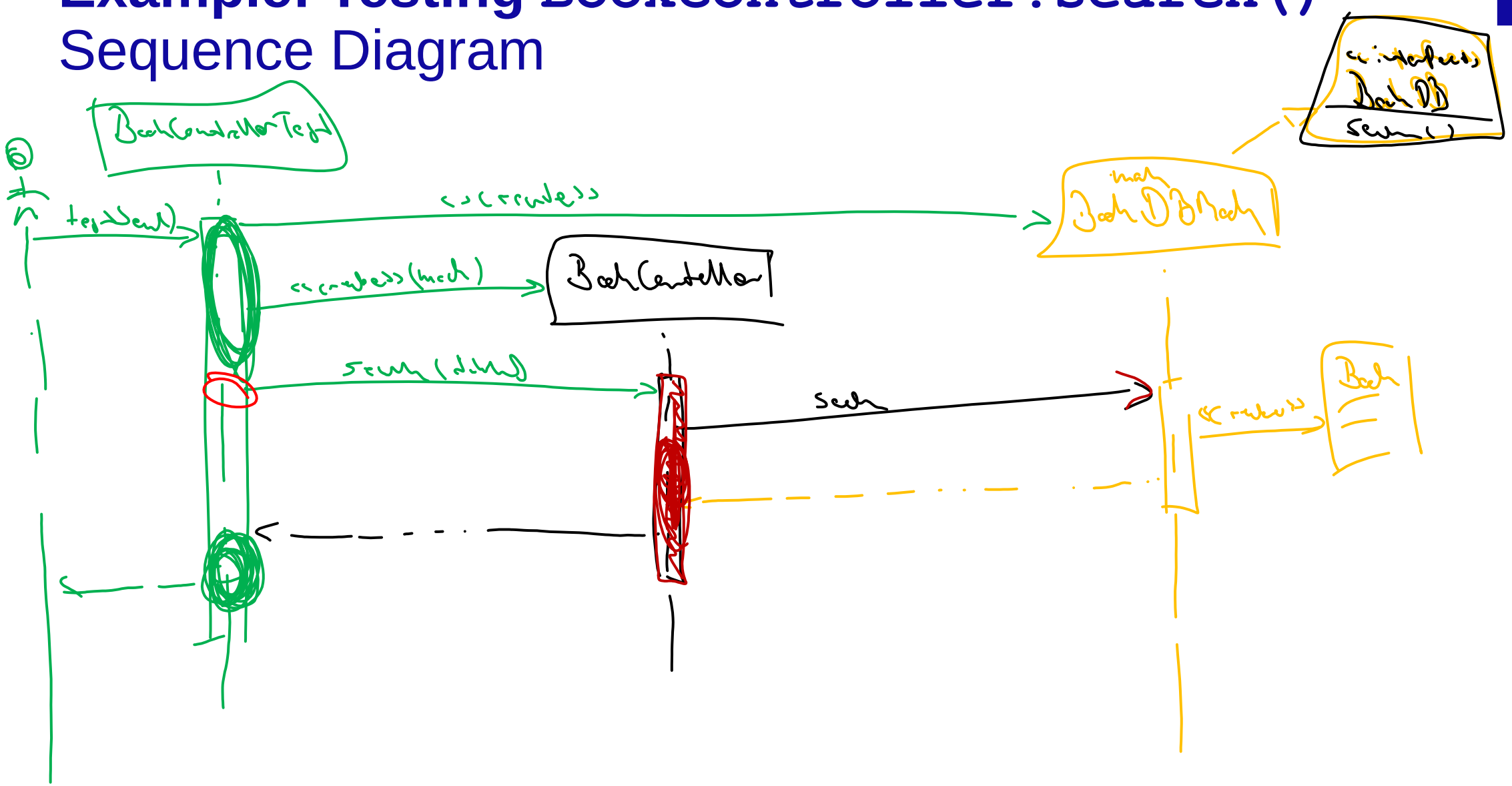
Recap: Library Search Sequence Diagram



Example: Testing `BookController.search()`
Class Diagram



Example: Testing `BookController.search()`
Sequence Diagram



Pitfalls

- Never refer to mock objects in production code, only in test fixtures
  - Use «interface»s so production code can remain agnostic of mock objects
- Be aware of what you are coding, mocking and testing
  - Make sure you are not testing your mock objects, or your test fixtures
  - Remain aware of what is still to be implemented
- Be aware of testing economics
  - What is worth testing? What specification do you want to enforce? What defects can occur?
  - Focus tests foremost on the specification; less on exotic error conditions you can imagine
- Be aware that your tests imply certain solutions
  - Consider if those solutions are really optimal
- Make sure all your tests are independent
  - Tests should not be influenced by / not rely on side effects from previous tests

Thank you!

book@hi.is