Unit Testing

Non-Business

Agenda

Demo Project (Maven-built)

Unit Tests

Exercise 1

Lunch

Dependencies

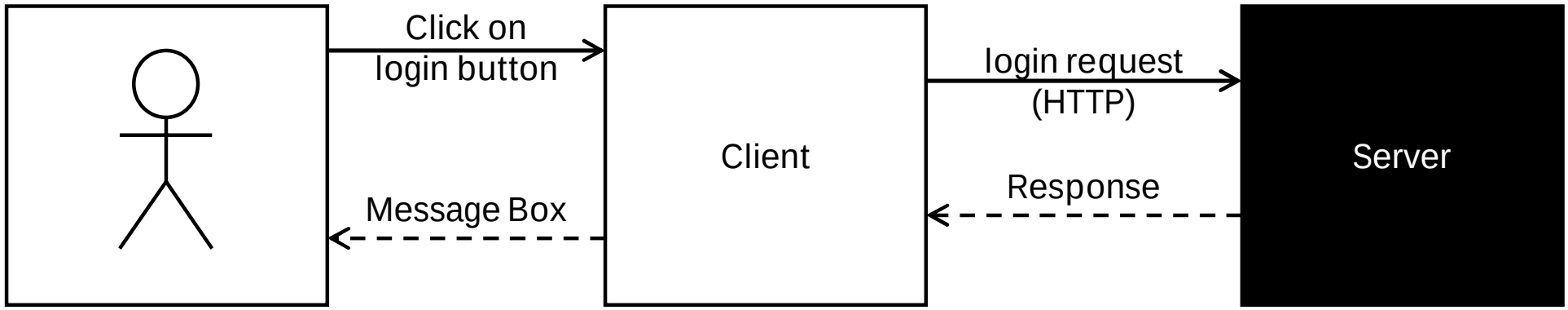
Exercise 2

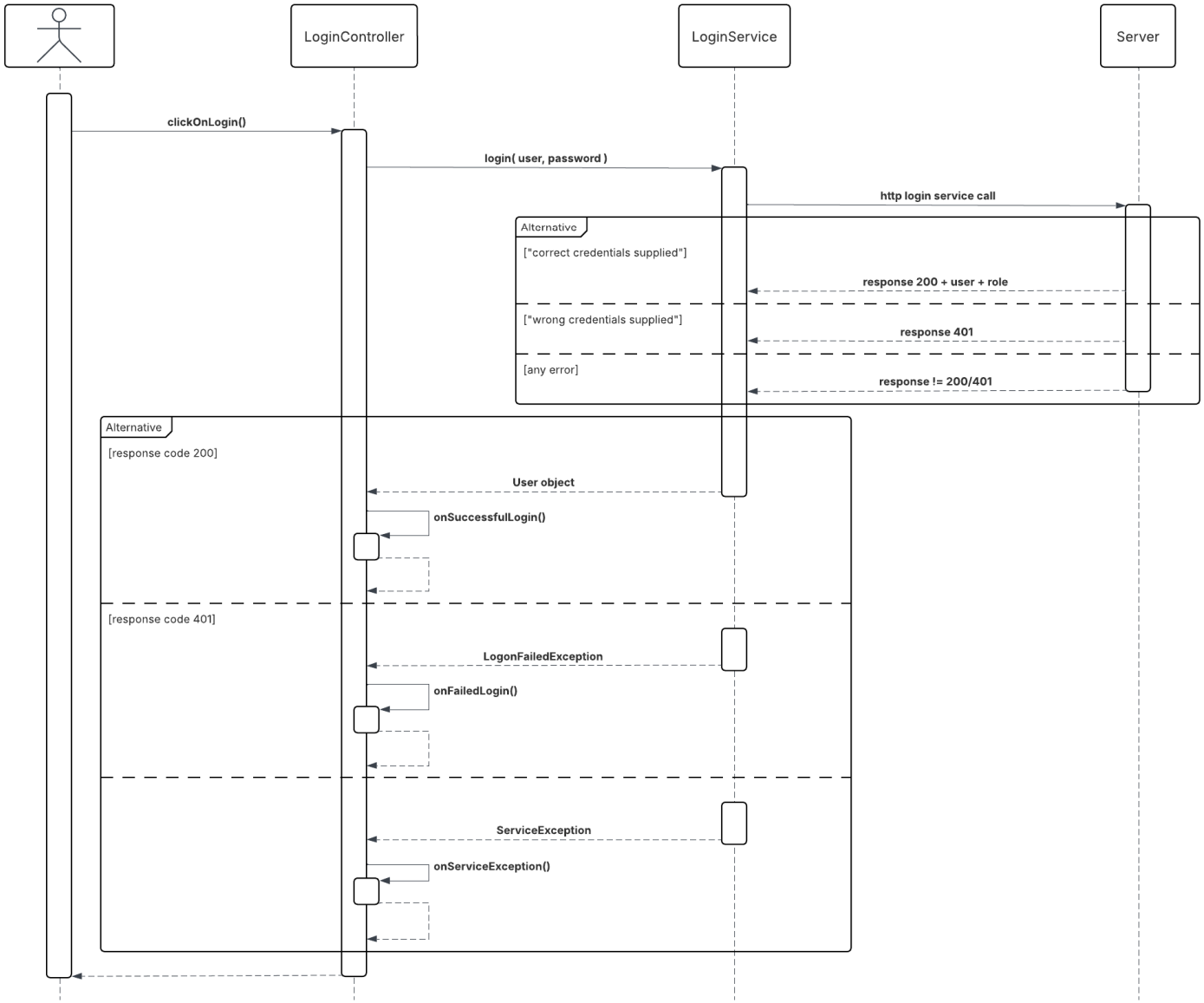
Demo Project

https://github.com/PatrickATdIT/atdit-y2025-a

Non-Business

Functionality




LoginController
LoginService
Server
clickOnLogin()
login(user, password)
http login service call
Alternative
["correct credentials supplied"]
response 200 + user + role
["wrong credentials supplied"]
response 401
[any error]
response != 200/401
Alternative
[response code 200]
User object
onSuccessfulLogin()
[response code 401]
LogonFailedException
onFailedLogin()
ServiceException
onServiceException()

Maven

Build Tool

Automates the whole build process, e.g.

- Compile
- Package
- Test
- Deploy

Dependency Management

- Automatic import and update of external libraries
- Default source Maven central repository; alternatives possible

Already integrated in many IDEs

Maven

Central configuration file

- pom.xml
- One per module

Convention over configuration

- Standardised project structure
- Standardised build cycle
- Plugins can be used to extend build process and folder structure

Unit Tests

Non-Business

What is a test in development context?

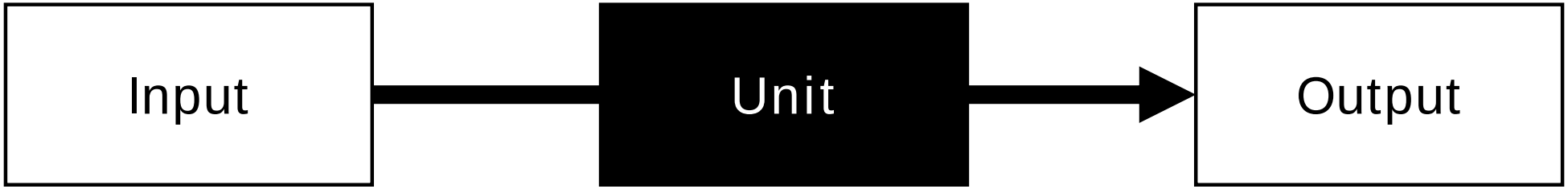
A test is a manual or automated procedure used to evaluate and verify the correctness, functionality, and performance of software under specific conditions.

What is a unit test?

A unit test is a type of automated test designed to verify that individual components or methods of a software program work as intended in isolation from the rest of the system.

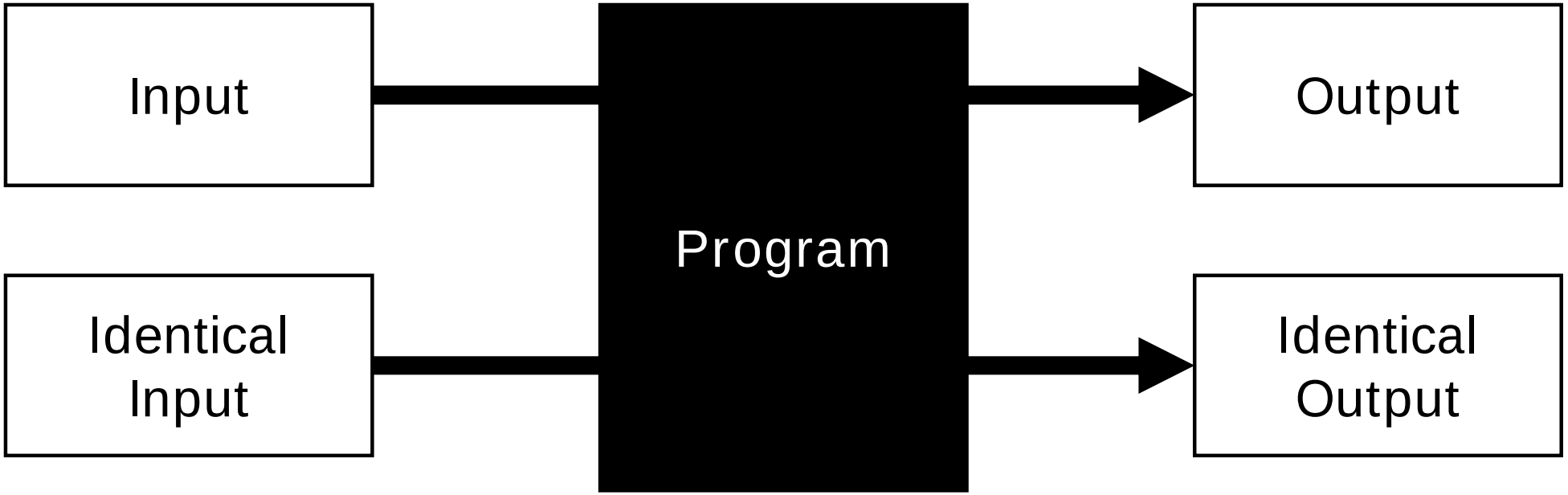
Priciple

Program units process data (input) to create some result (output)



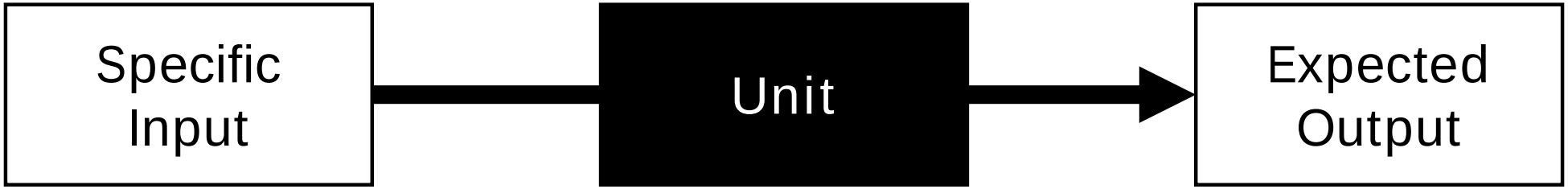
Priciple

Results must be reproducible



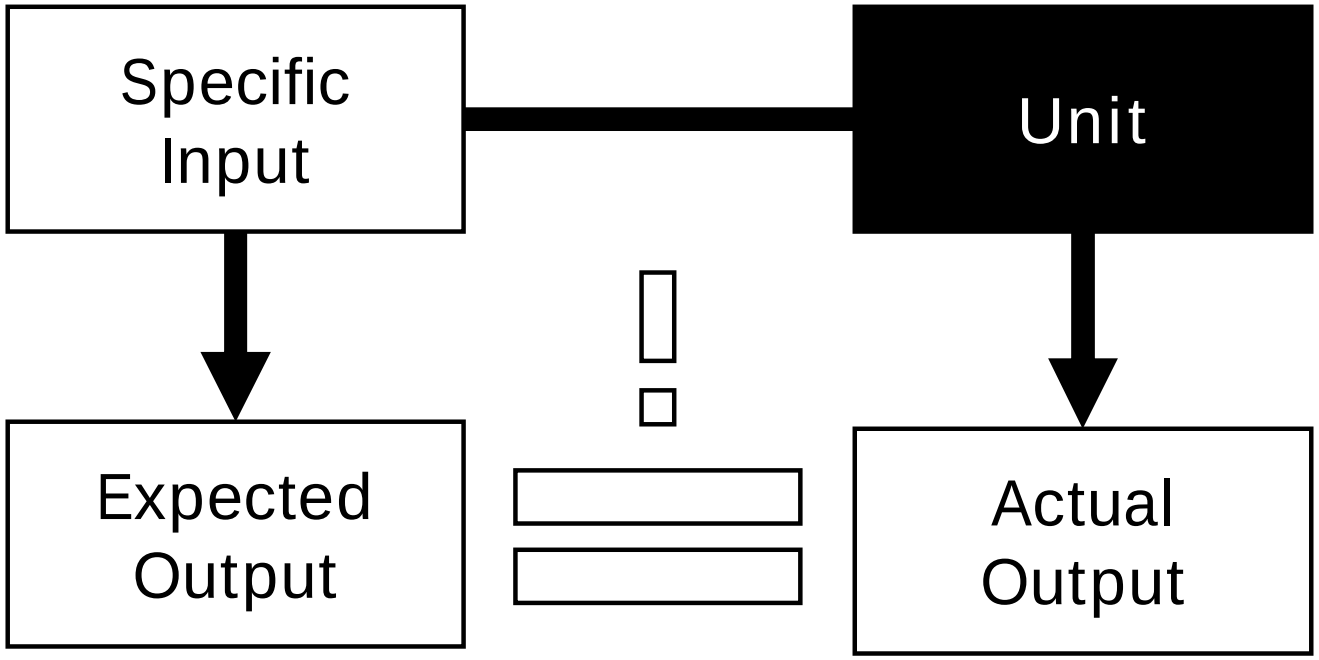
Priciple

Program units are deterministic



Priciple

Expected and actual output can be compared automatically



Unit Tests

Automated

- Can run automated
- Results can be interpreted automated

Deterministic

- Results are predictable
- Certain input creates certain output

Reproducible

- Results of the same unit tests will not change
- Independent of sequence or number of executions

Unit Tests

Isolated

- Each unit test tests only one single module
- There must be no dependencies within the test

Simple

- Unit test code should be easy to read and understand
- A single test should only test one single aspect of a module

Contract based

- Unit tests should focus on testing interfaces
- Unit tests should not test for internals of the module
- Regard coverage anyway

Why Unit Testing?

Error Detection

- Errors are detected during development, even before the actual testing phase

Code Quality

- Unit tests require testable code
- Testable code produces better code quality
  - Modularized
  - Small program units
  - SOLID by design
  - Promotes clean code

Software Quality

- Code quality is part of software quality
- Tests can be conducted faster, more frequently, and in greater detail (compared to manual testing)

Why Unit Testing?

Promotes Agility

- Avoidance of regression errors
- Applies to development and maintenance

Faster Error Analysis

- Maintainable Code
- Failing unit tests directly point to the faulty module
- The cause of failing unit tests is always the most recent code change

Documentation

- Unit tests are small and easy to understand
- Intended functionality can be derived from input and expectations
- Intended interface usage can be derived
- Better than nothing!

Demo 1

https://github.com/PatrickATdIT/atdit-y2025-a

Non-Business

Excercise 1

https://github.com/PatrickATdIT/atdit-y2025-a

https://github.com/PatrickATdIT/atdit-y2025-b

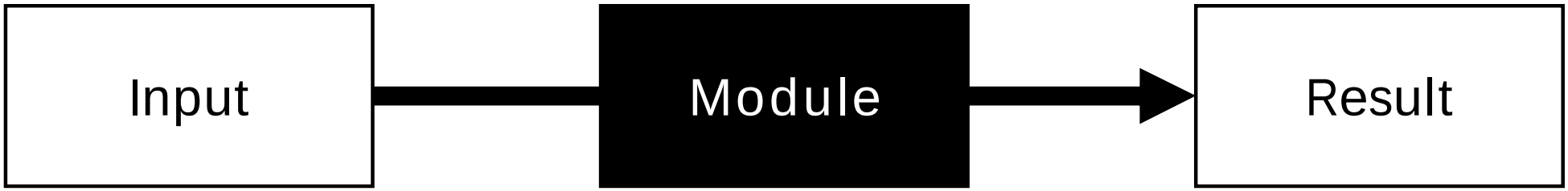
Non-Business

Dependencies

Non-Business

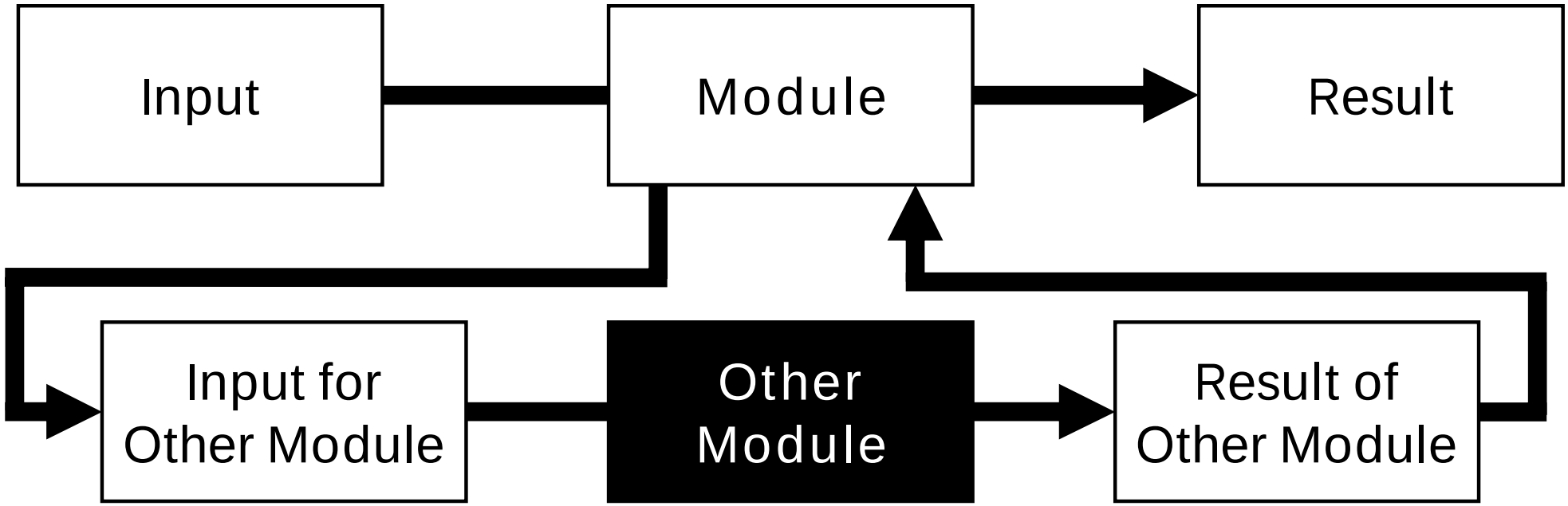
Remember?

Unit Tests must test single modules in isolation



In Reality

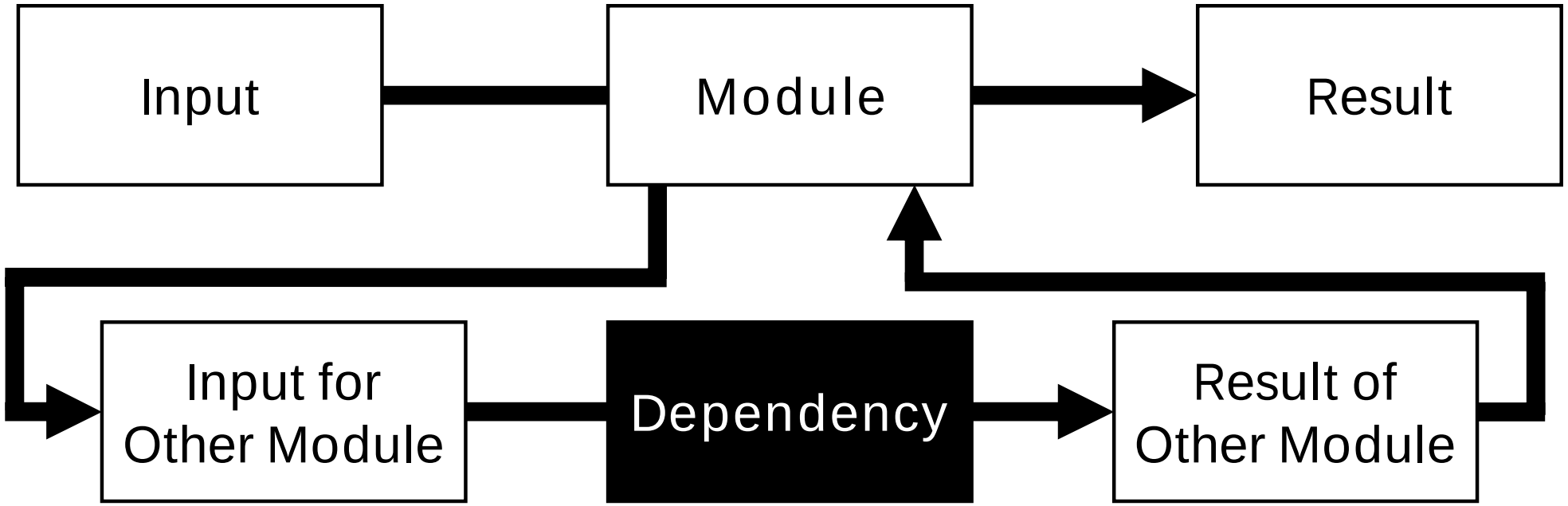
Modules usually use other modules



Consequence

Other modules can influence the results of our module

Our module depends on the other module; the other module is a dependency

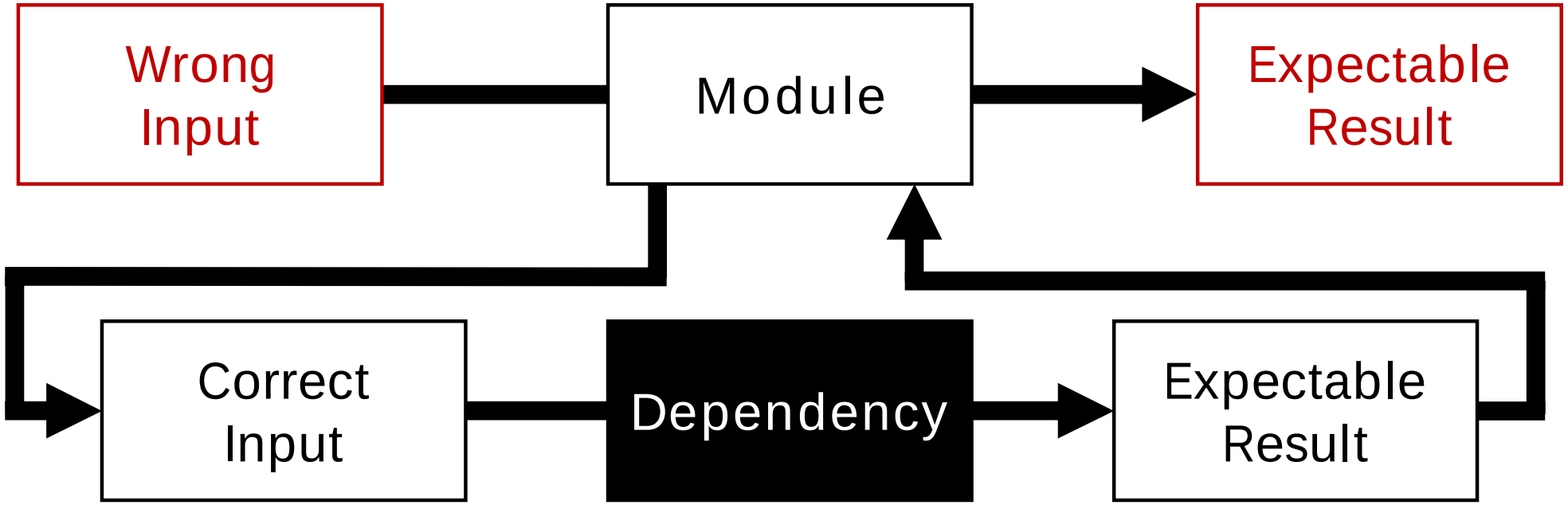


Sources of Errors

Consumer provides wrong input

Own unit tests are green and confirm our module works fine

Error is with the consumer (wrong input)

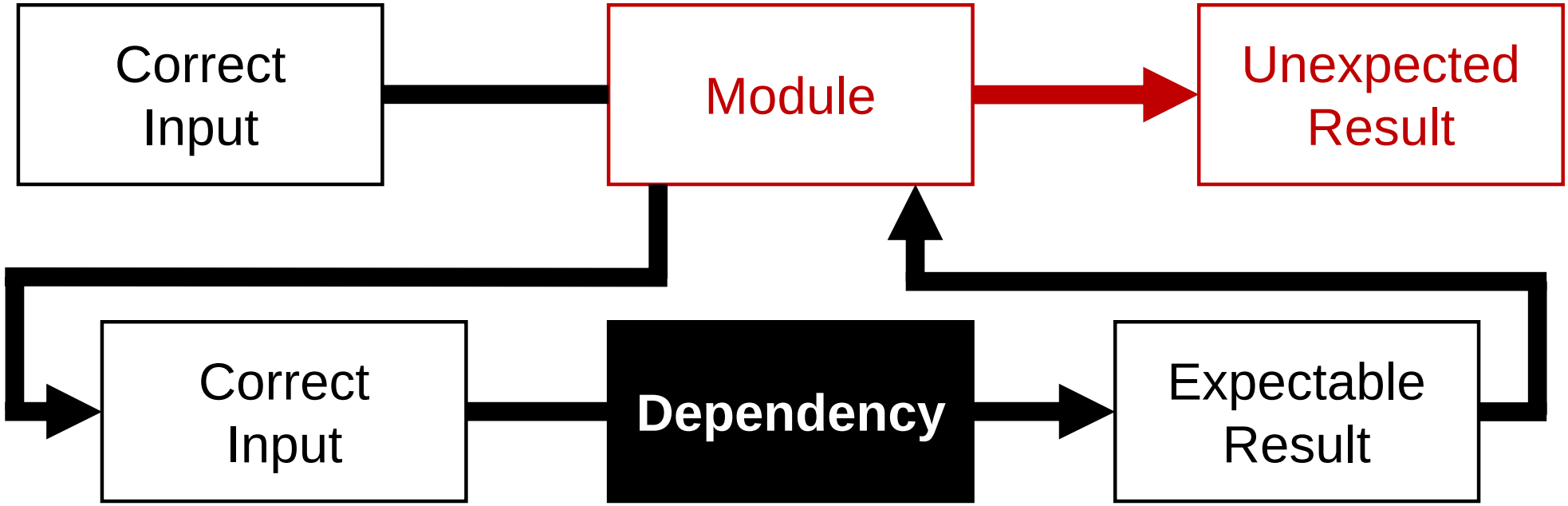


Sources of Errors

Our module has a bug after the dependency call

Our unit tests fail, proofing that our module has a bug

Error is with our module

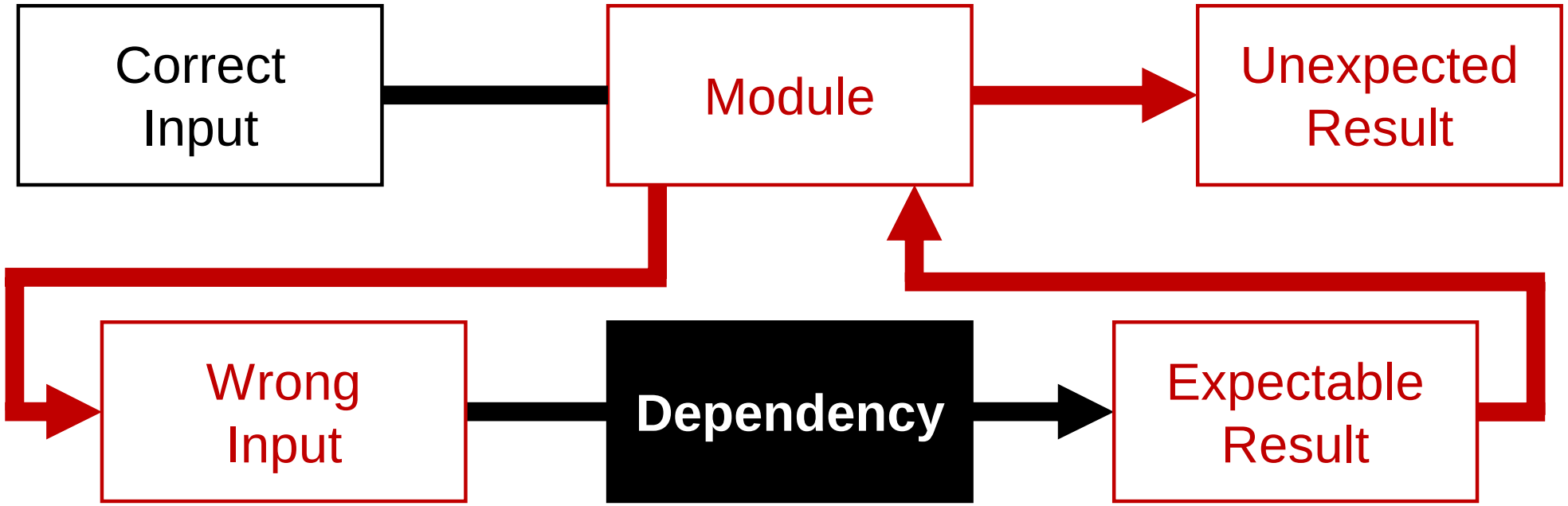


Sources of Errors

Our module has a bug that creates wrong input for the dependency

Unit tests fail, proofing we have a bug in our module

Responsibility is with us

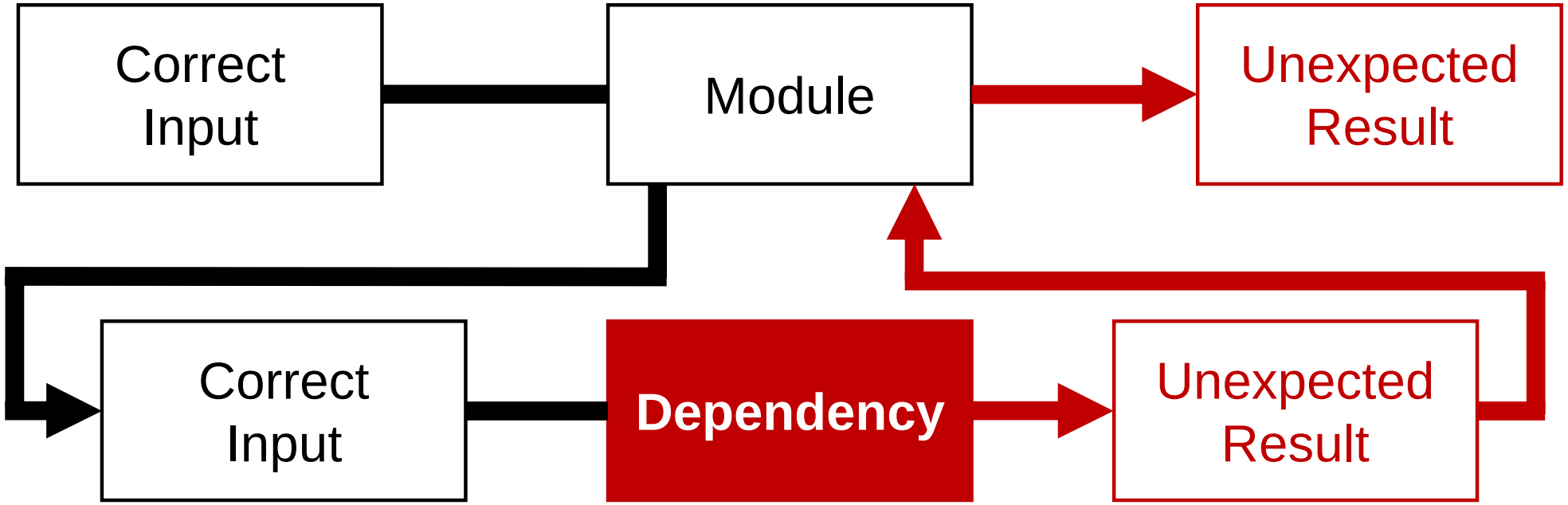


Sources of Errors

There is a bug in the dependency

Unit tests go red indicating our module has a bug

The actual error is within the dependency, but blame goes to us due to red unit tests



Dependencies must be resolved

In unit testing, dependencies must be eliminated; they must not influence the own tests

Reason: Dependencies are neither within our control nor responsiblity

Target is to replace them by so called mocks, who pretend to be the real dependency

Reason: If we create the mock, we are in control of ist run time behavior

Precondition: testable code

Testable code

The dependency must implement an interface (supplier responsibility)

If the dependency does not implement an interface, we can wrap it in an own interface definition

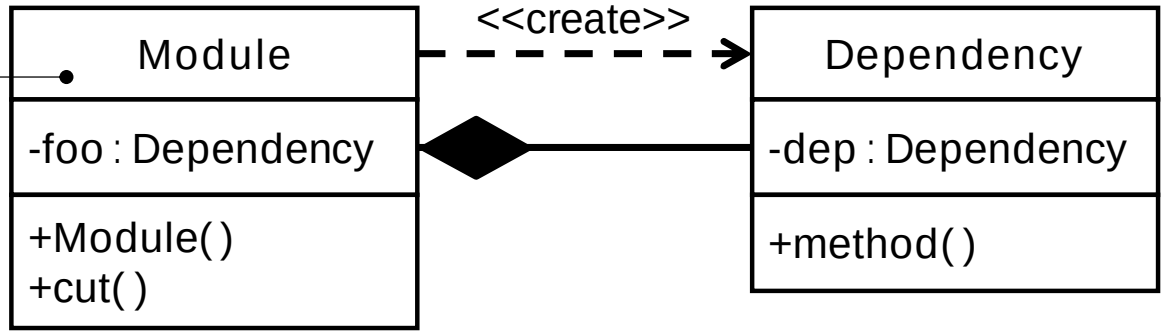
Unit tests must be able to replace the implementation of the dependency (dependency injection), this is in own responsibility

Typical methods of dependency injections:

- Constructor parameters
- Parameter of code-under-test (cut) method
- Inheritance (overriden supplier methods, but not always possible)
- Public attribute within the CUT class (not recommended, breaks encapsulation)
- Instantiation by factory (this only moves the responsibility to the factory)
- Reflexion (Beware: Could be prohibitted, e.g. per JPMS module definition)
- Friendship (Classes can access their private parts; not supported by Java)

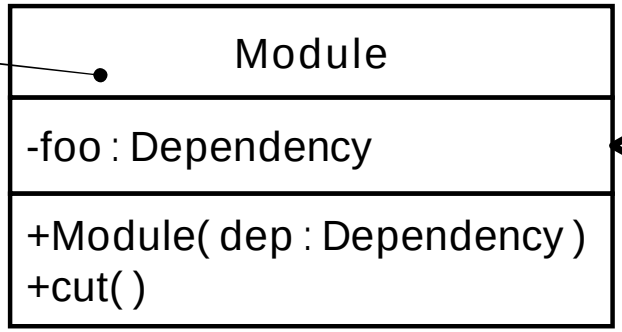
Not testable code

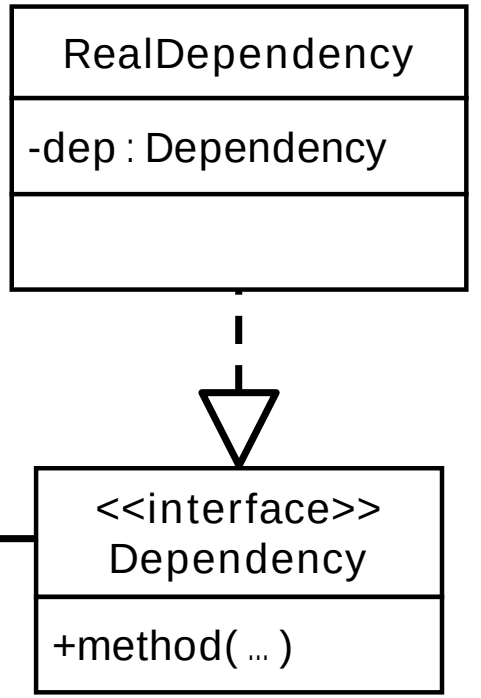
```
public class Module {
  public Module( ) {
    dep = new Dependency ( );
  }
  public void cut( ) {
    dep.method( );
  }
}
```



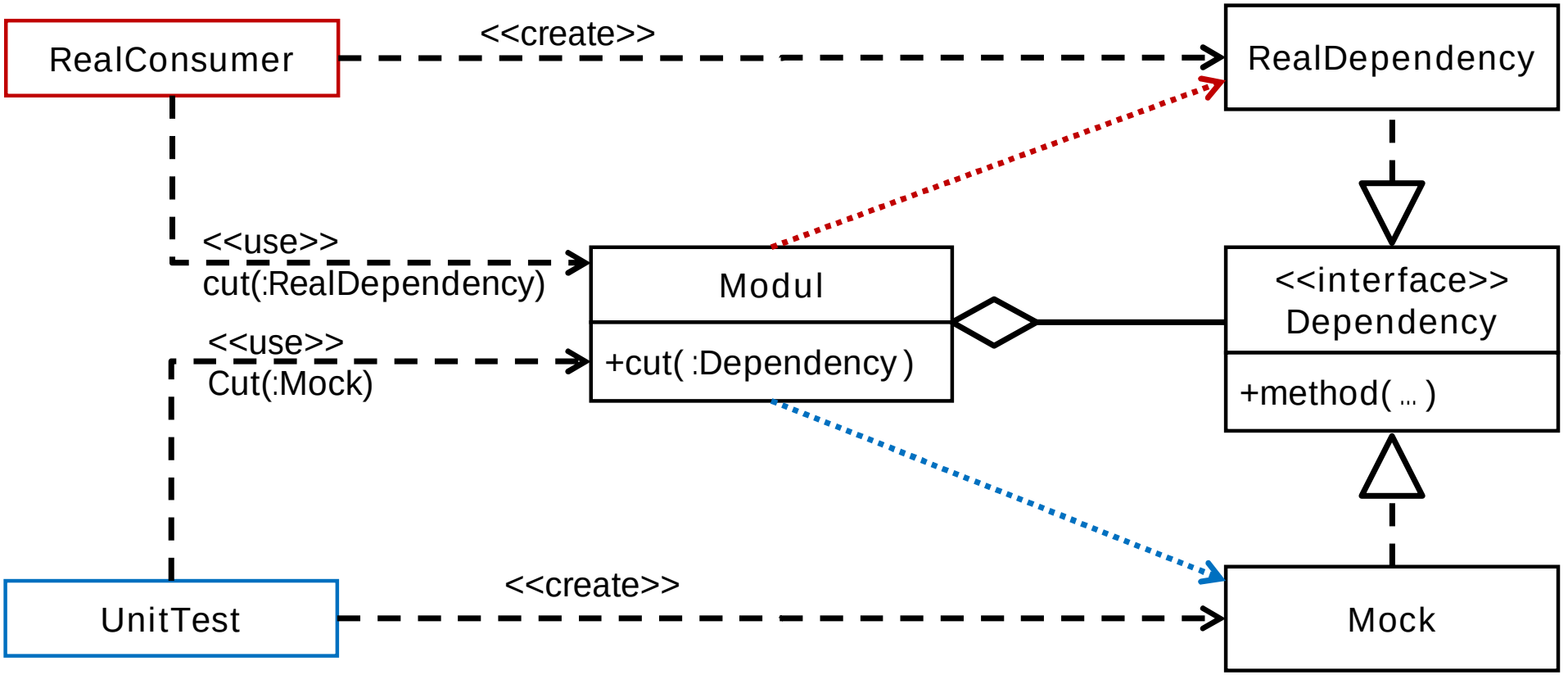
Testable code (constructor injection)

```
public class Module {
  public Module(Dependency dep )
{
    this.dep = dep;
  }
  public void cut( ) {
    dep.method( );
  }
}
```





Resolve dependency (method injection)

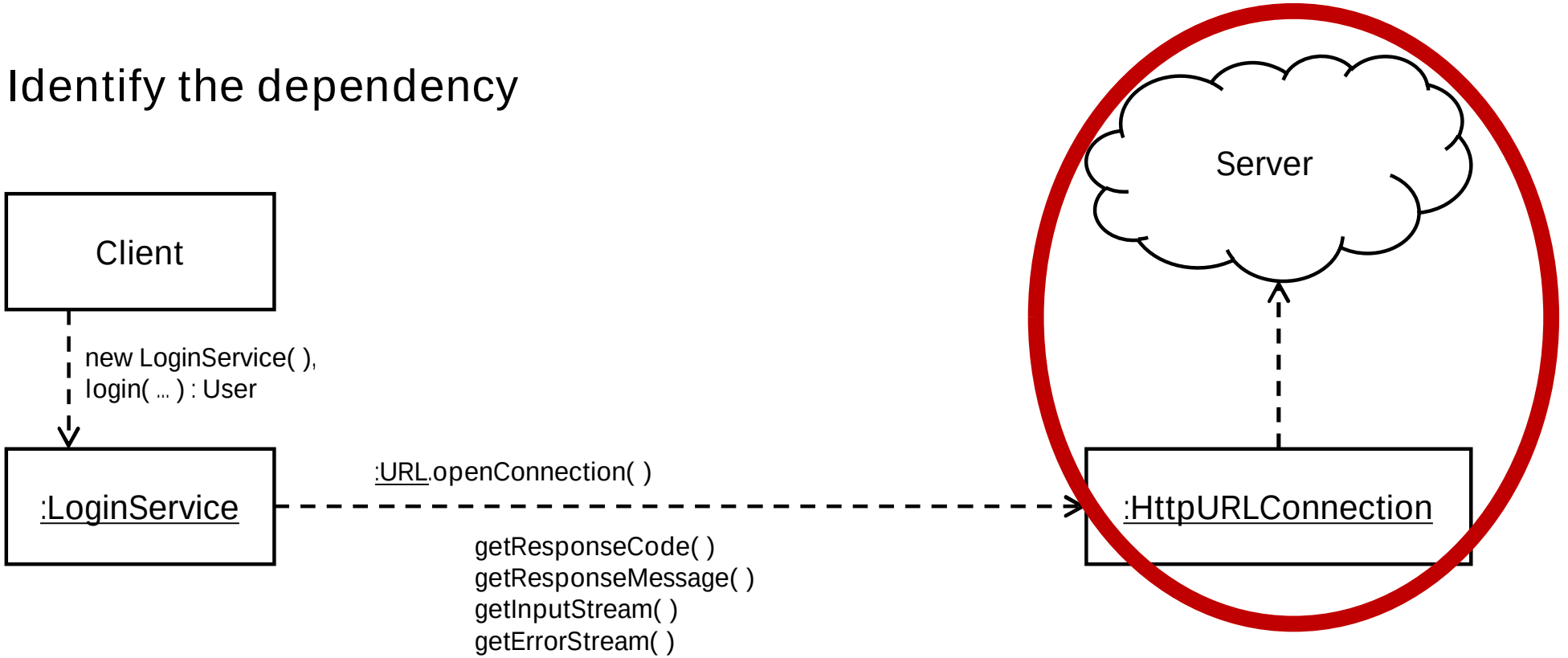


Demo 2

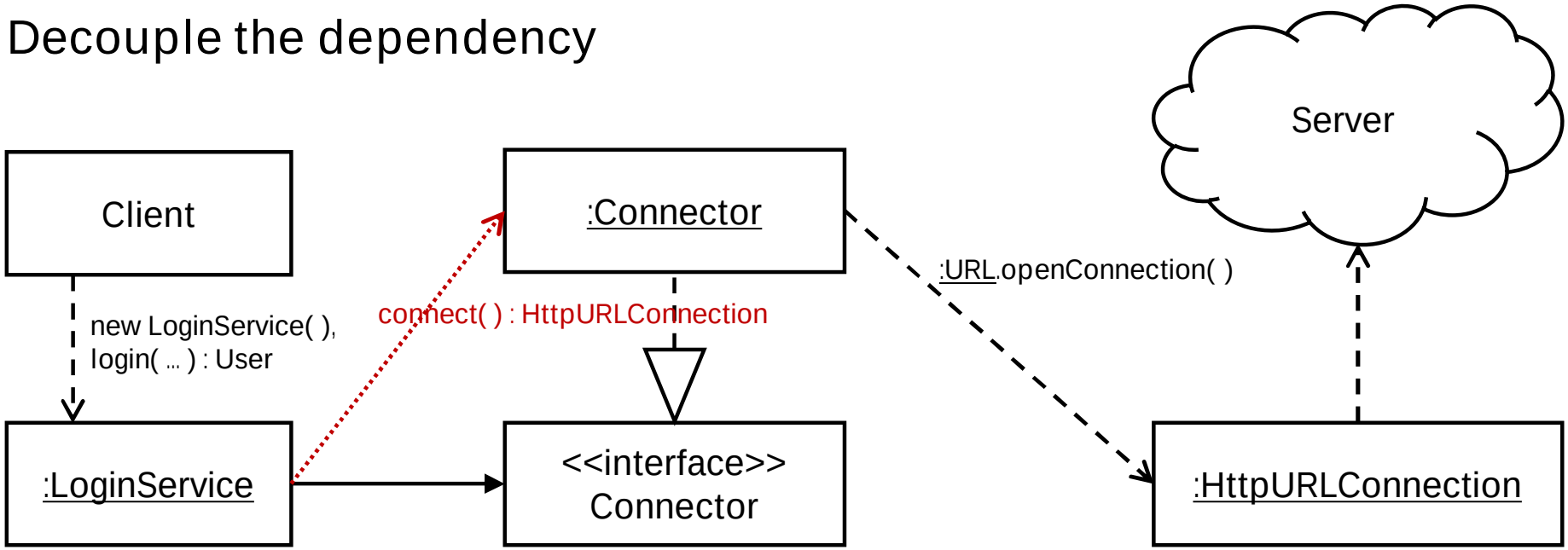
https://github.com/PatrickATdIT/atdit-y2025-b

Non-Business

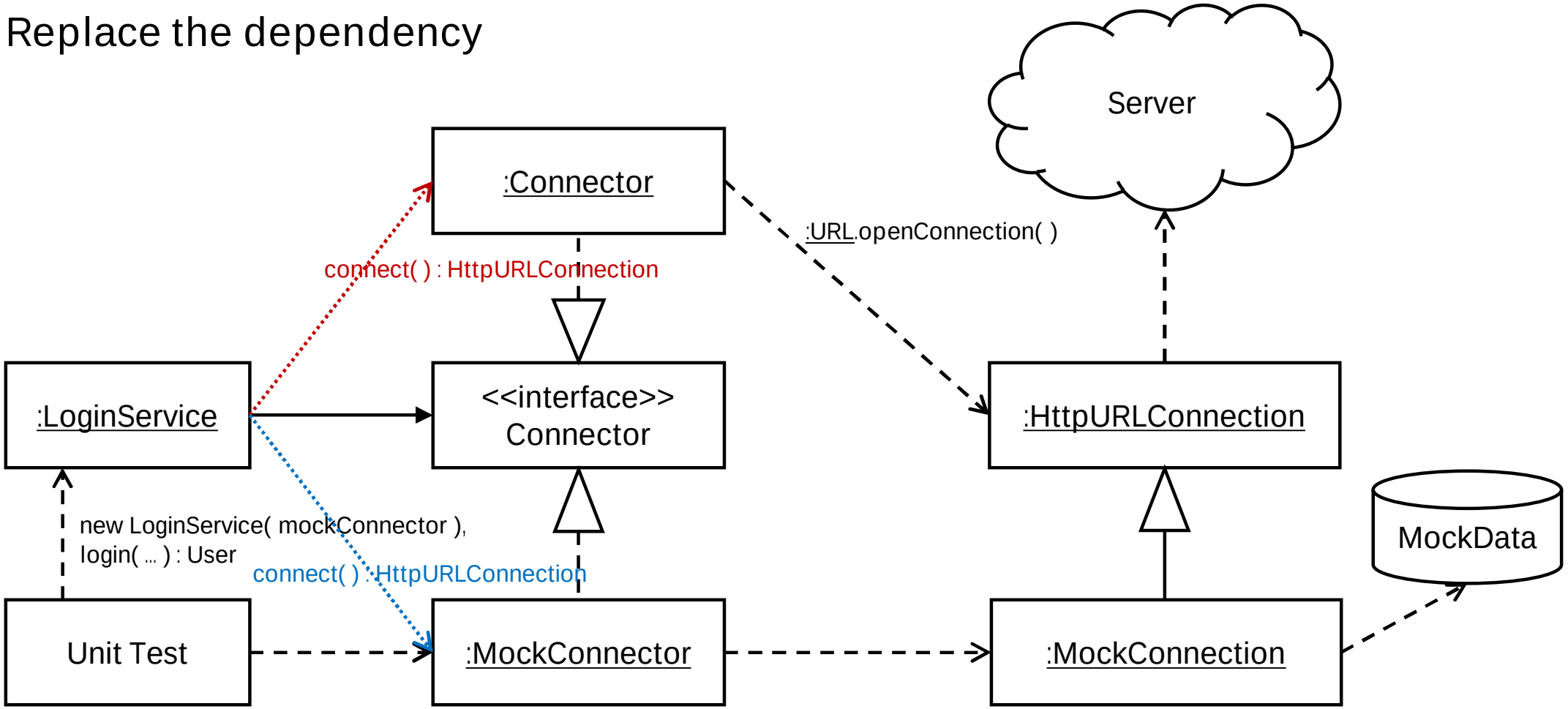
Identify the dependency



Decouple the dependency



Replace the dependency



Exercise 2

https://github.com/PatrickATdIT/atdit-y2025-b

https://github.com/PatrickATdIT/atdit-y2025-c

Non-Business

Further Reading

Non-Business

Further Reading

Principles of object oriented design (German)

SOLID-Concepts (German)

Robert C. Martin - Clean Code (Book, English)

Maven (English)

JUnit 5 (English)

Mockito (English)