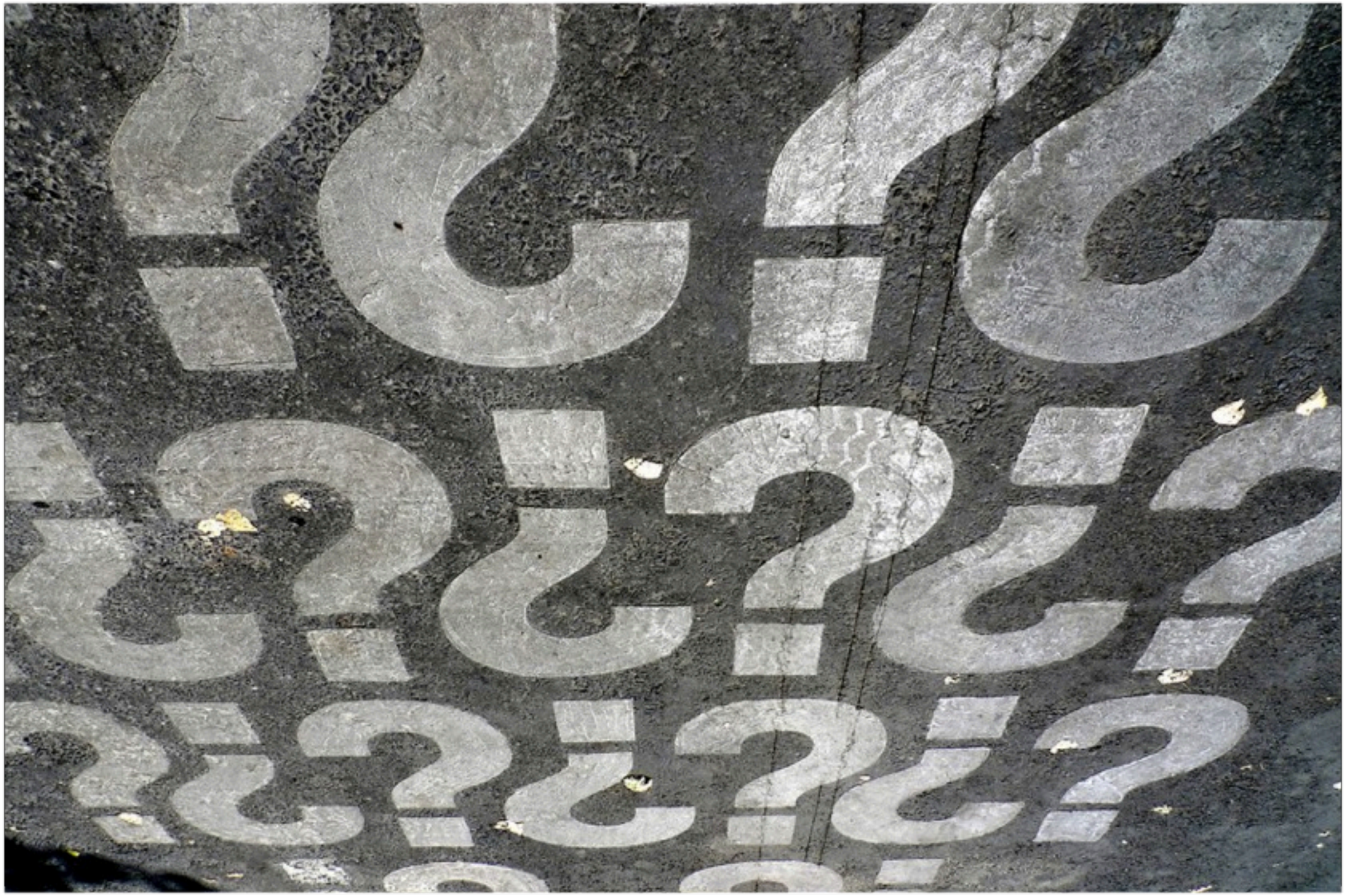
SCAFFOLDING WITH JMOCK

Software Engineering Class

Prof. Adriano Peron

June 6, 2013

Valerio Maggio, Ph.D.
valerio.maggio@unina.it



EXERCISE 1

Calculator

A **BIG** THANK YOU GOES TO..

Luciano Conte

Vittorio Parrella

Marco Zeuli

CALCULATOR CLASS

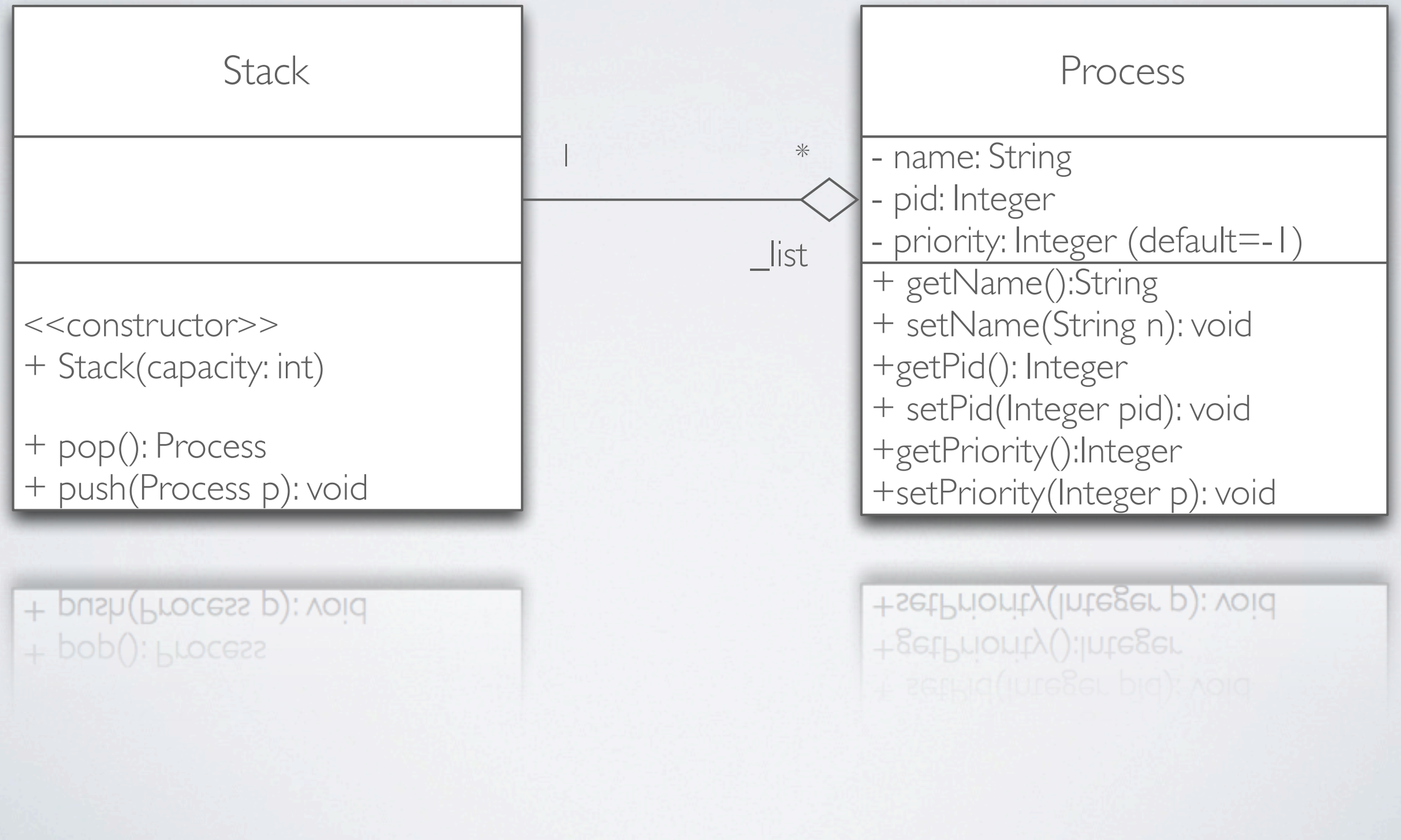
- **Requirements:**
  - Input numbers cannot have more than 5 digits;
  - The calculator can remember a given (unique) number;
  - Only non-negative numbers are allowed.
  - In case of negative numbers, an exception is thrown!

Calculator
- memory: double + MAX_DIGITS_LEN: int = 5 <<final>> <<static>>
+ add (double augend, double addend): double + subtract (double minuend, double subtrahend): double + multiply (double multiplicand, double multiplier): double + divide (double dividend, double divisor): double + addToMemory(double number): void + recallNumber(): double

EXERCISE 11

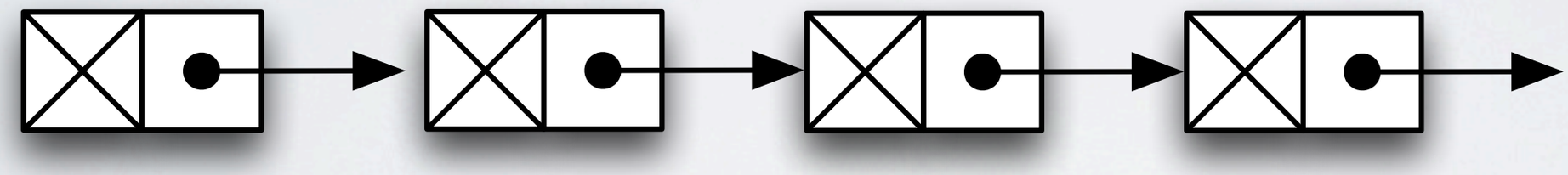
Stack

STACK: **LIFO** QUEUE



BRIEF RECAP OF: "PROGRAMMING CLASS"

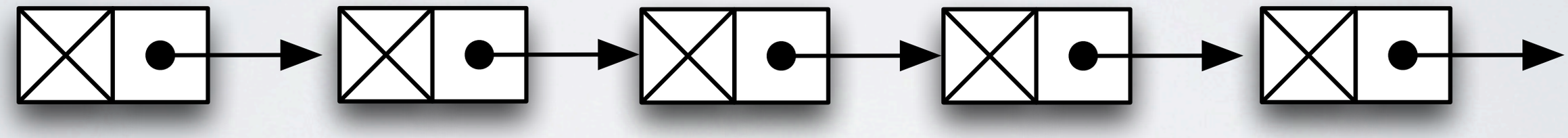
FIFO QUEUE



enqueue()

BRIEF RECAP OF: "PROGRAMMING CLASS"

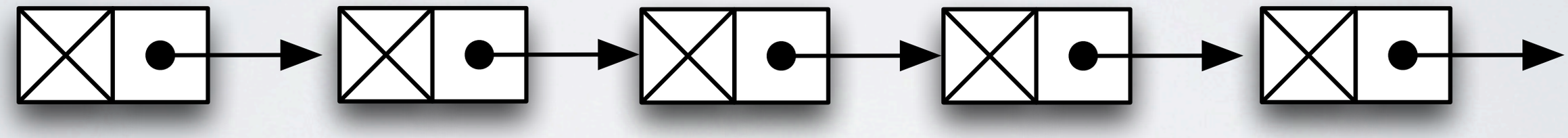
FIFO QUEUE



enqueue()

BRIEF RECAP OF: "PROGRAMMING CLASS"

FIFO QUEUE

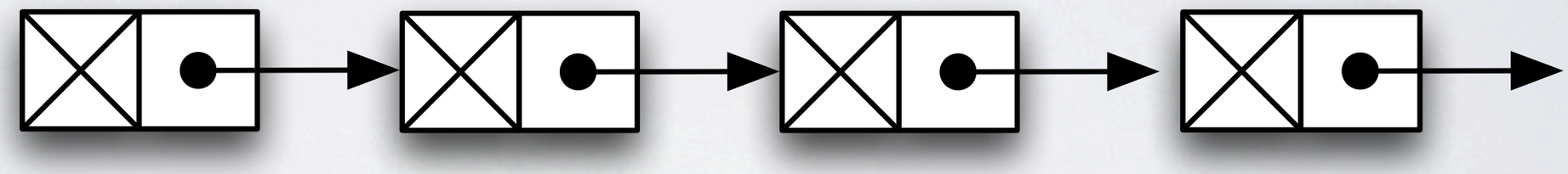


enqueue()

dequeue()

BRIEF RECAP OF: "PROGRAMMING CLASS"

FIFO QUEUE

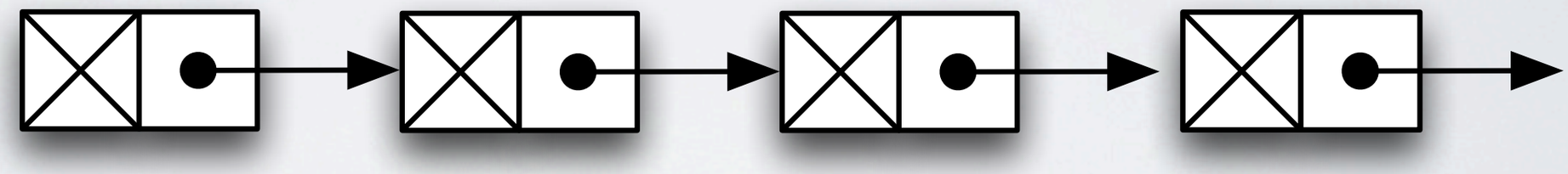


enqueue()

dequeue()

BRIEF RECAP OF: "PROGRAMMING CLASS"

LIFO QUEUE

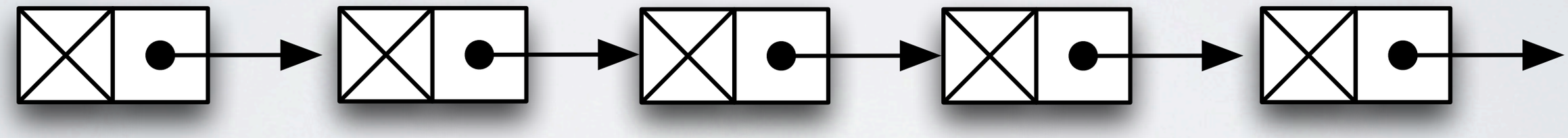


enqueue()

dequeue()

BRIEF RECAP OF: "PROGRAMMING CLASS"

LIFO QUEUE

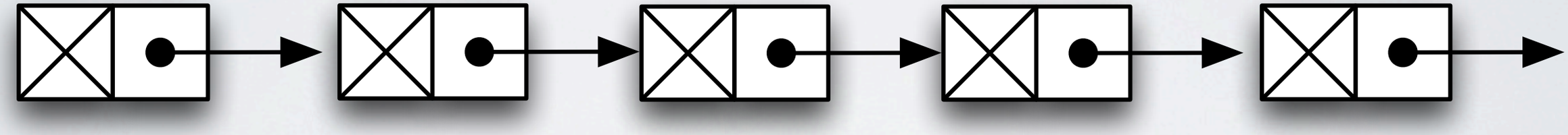


enqueue()

dequeue()

BRIEF RECAP OF: "PROGRAMMING CLASS"

LIFO QUEUE

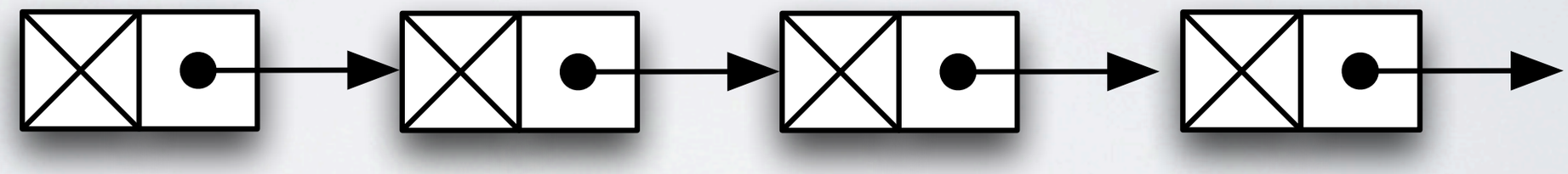


enqueue()

dequeue()

BRIEF RECAP OF: "PROGRAMMING CLASS"

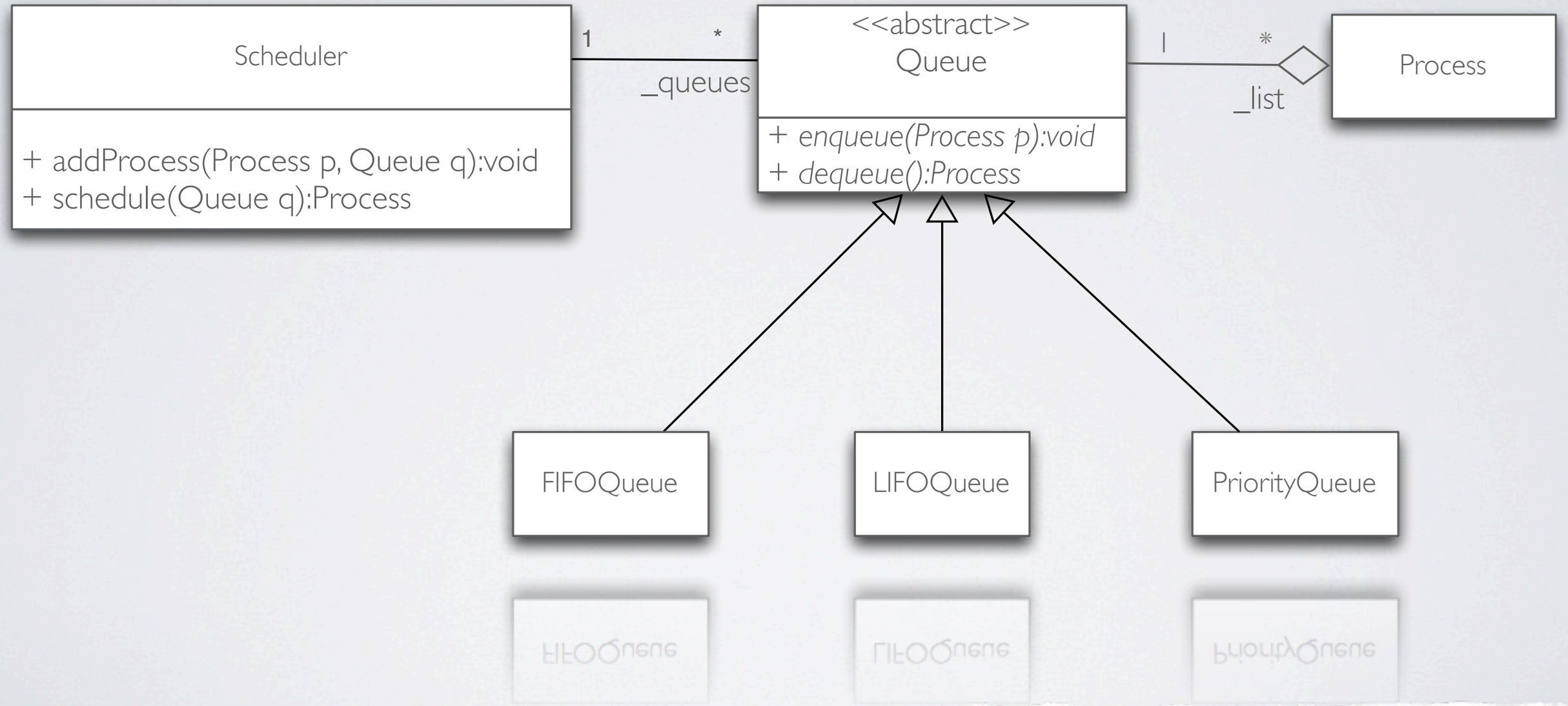
LIFO QUEUE



enqueue()

dequeue()

DO DON'T TRY THIS AT HOME!



Q: How would you **test** Scheduler?

Remember: Unit tests run in **isolation!**

TEST SCAFFOLDING

INTEGRATION TESTING

```
public class TestUserAccount {

    private Connection dbConnection;

    @Before public void
    setUp(){
        this.dbConnection = new dbConnection("...");
        this.dbConnection.connect();
    }

    @Test public void verifyAccountCredentials(){
        //....
    }

    @After public void
    tearDown(){
        this.dbConnection.close();
        this.dbConnection = nulll;
    }
}
```

INTEGRATION TESTING

```
public class TestUserAccount {

    private Connection dbConnection;

    @Before public void
    setUp(){
        this.dbConnection = new dbConnection("...");
        this.dbConnection.connect();
    }

    @Test public void verifyAccountCredentials(){
        //....
    }

    @After public void
    tearDown(){
        this.dbConnection.close();
        this.dbConnection = nulll;
    }
}
```

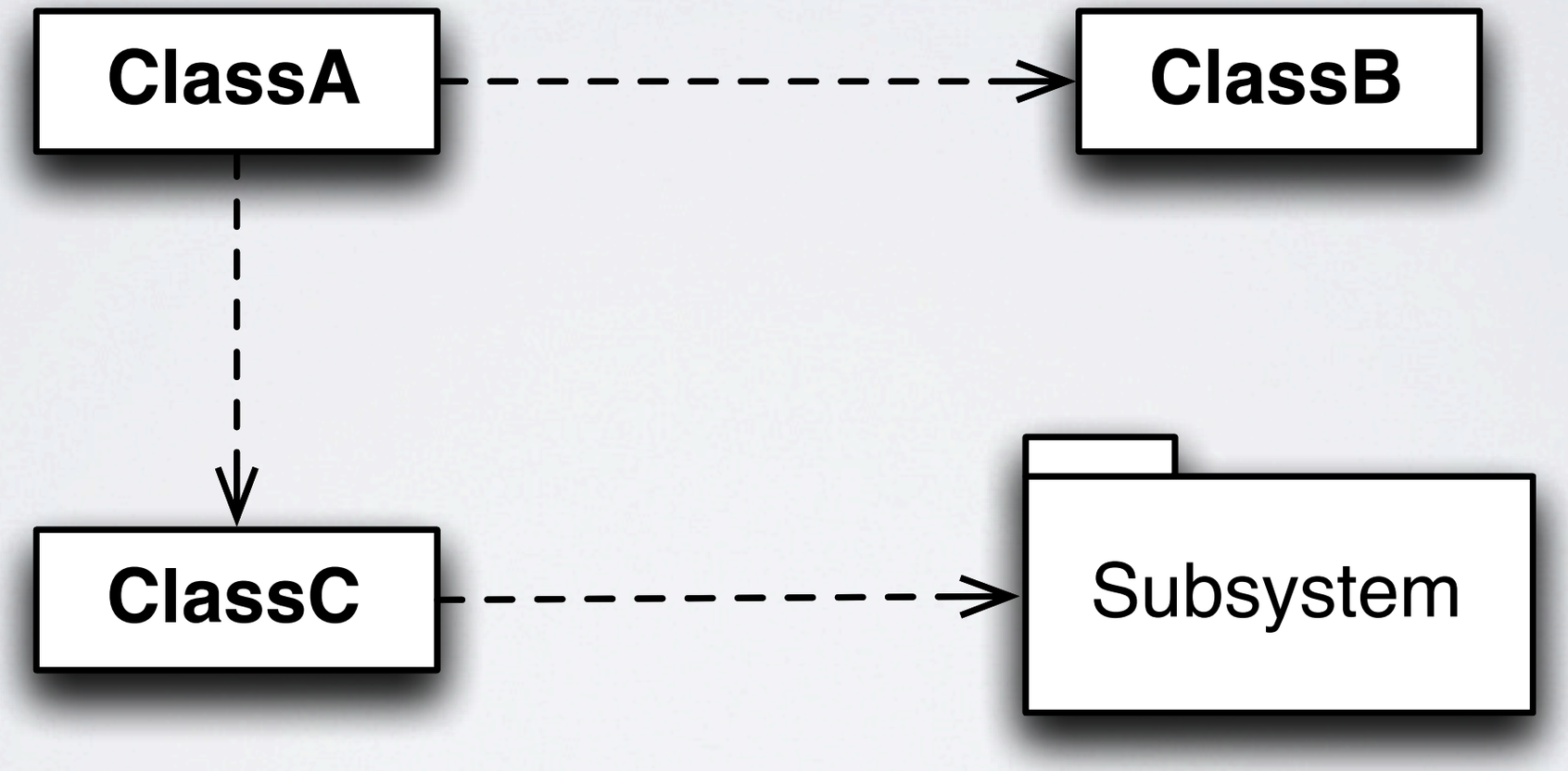
INTEGRATION TESTING PROBLEM

- Integrate multiple components implies to decide in which order classes and subsystems should be integrated and tested

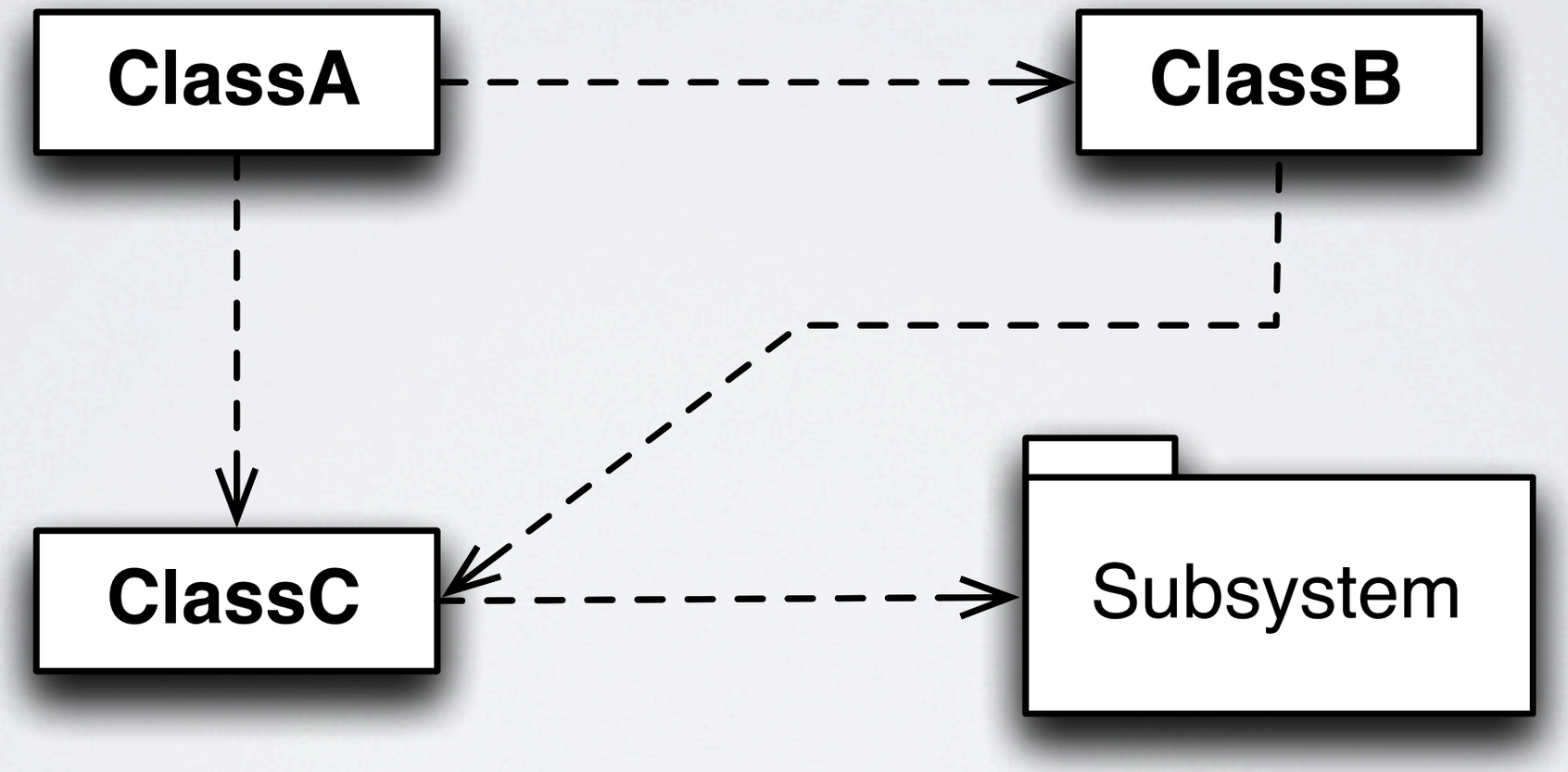
- CITO Problem
  - Class Integration Testing Order Problem

- Solution:
  - Topological sort of dependency graph

INTEGRATION TESTING EXAMPLE



INTEGRATION TESTING EXAMPLE



TESTING IN ISOLATION

Testing in Isolation **benefits!**

TESTING IN ISOLATION

Testing in Isolation **benefits!**

Test code that have not been written

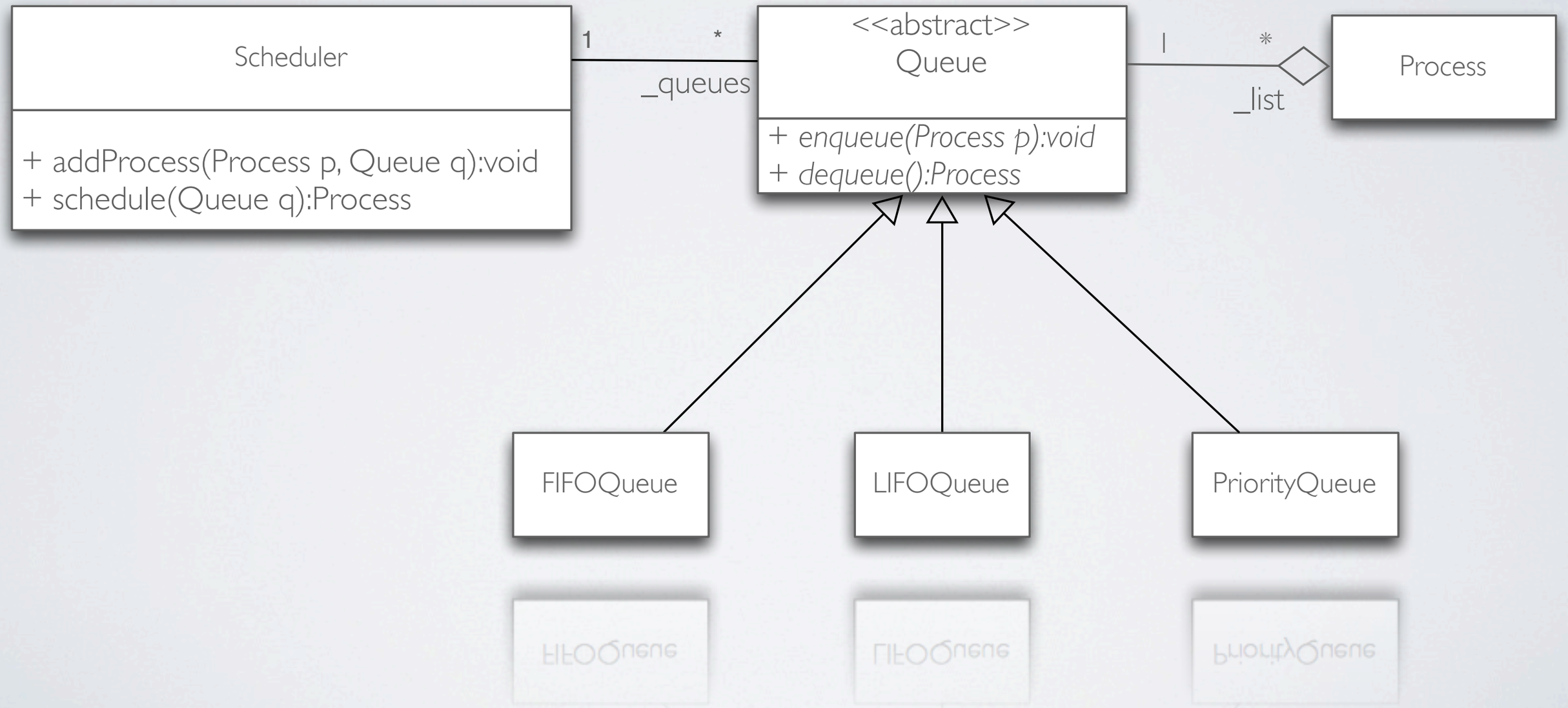
TESTING IN ISOLATION

Testing in Isolation **benefits!**

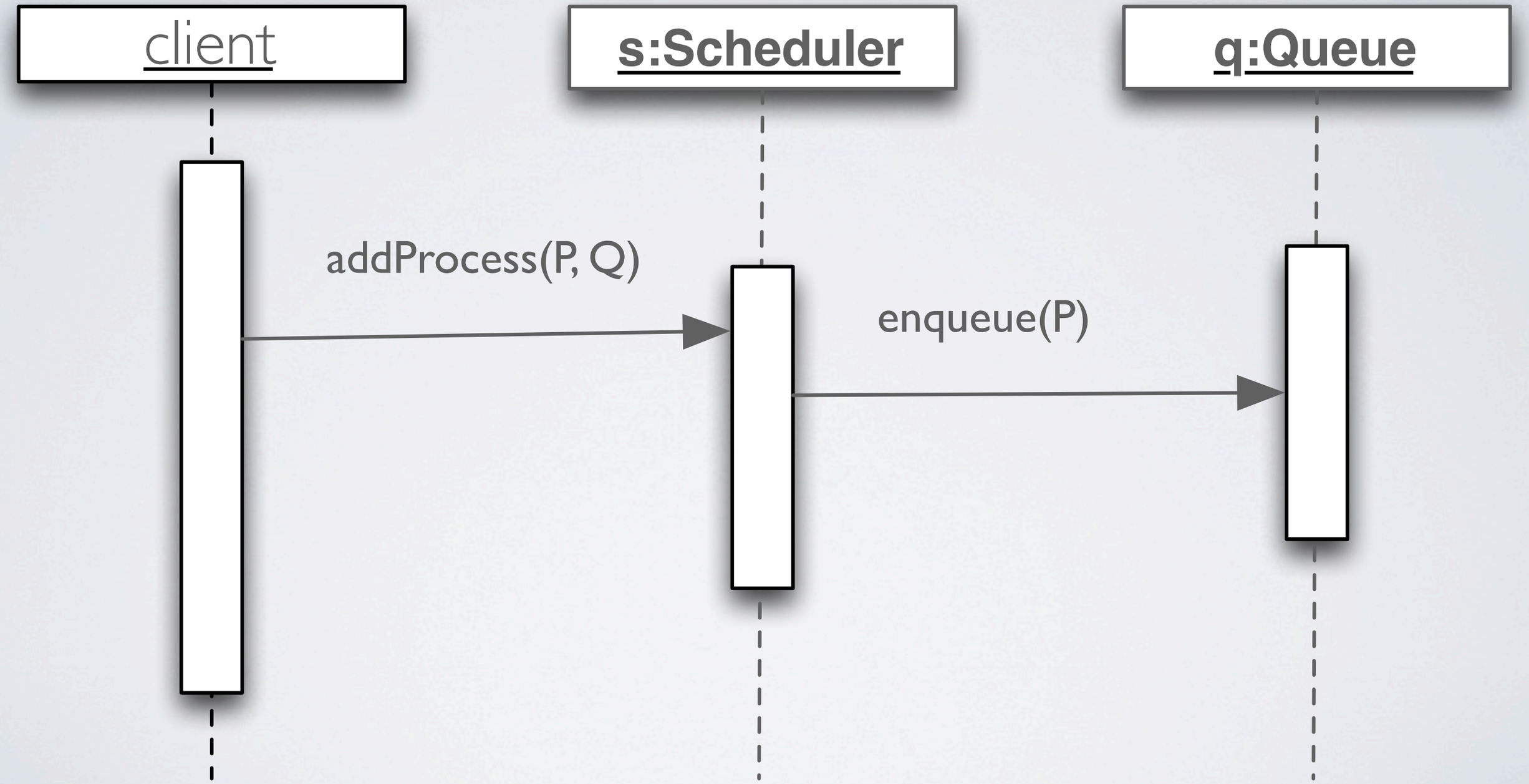
Test code that have not been written

Test only a single method (behavior) without side effects from other objects

SCHEDULER EXAMPLE



SCHEDULER@addProcess



SOLUTION WITH STUBS

```
public class DummyQueue implements Queue {

    @Override
    public void enqueue(Process p) {
        throw new RuntimeException();
    }
}

public class TestScheduler {

    @Test
    public void addProcessCallMethodEnqueueOfQueue() {
        Scheduler s = new Scheduler();
        try {
            DummyQueue q = new DummyQueue();
            s.addQueue(q);
            s.addProcess(new DummyProcess(), q);
            fail("addProcess did not call the enqueue method of queue");
        } catch (RuntimeException re) {}
    }
}
```

@Marco Zeuli

KEY IDEAS

- Wrap all the details of Code
  - (sort of) Simulation

- Mocks do not provide our own implementation of the components we'd like to swap in

- **Main Difference:**
  - Mocks test behavior and interactions between components

  - Stubs replace heavyweight process that are not relevant to a particular test with simple implementations

MOCK OBJECTS

- Powerful way to implement Behavior Verification
  - while avoiding Test Code Duplication between similar tests.

- It works by delegating the job of verifying the indirect outputs of the SUT

- Important Note: Design for Mockability
  - Dependency Injection Pattern

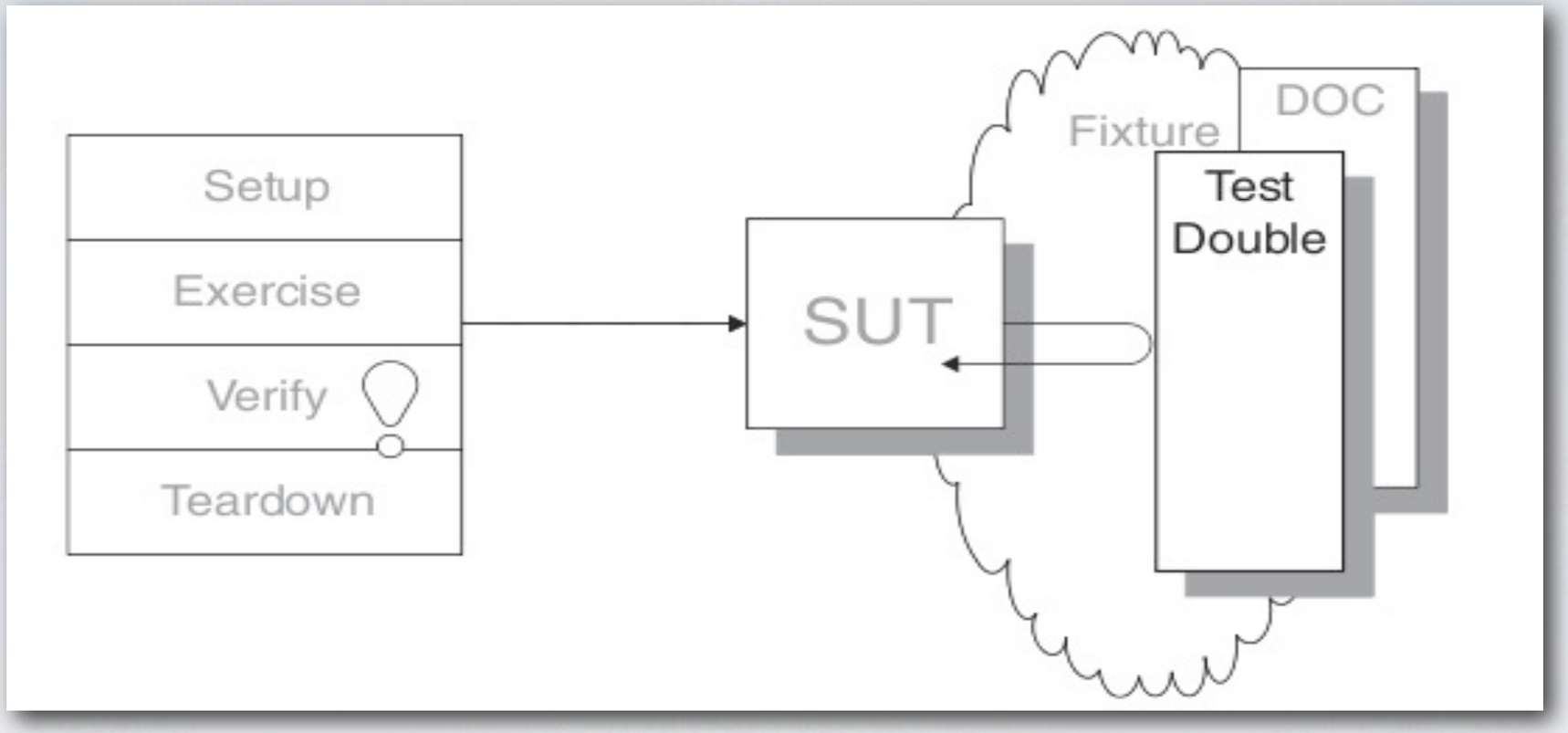
NAMING CONFUSION

- Unfortunately, while two components are quite distinct, they're used interchangeably.
  - Example: **spring-mock** package

- If we were to be stricter in terms of naming, stub objects defined previously are test doubles

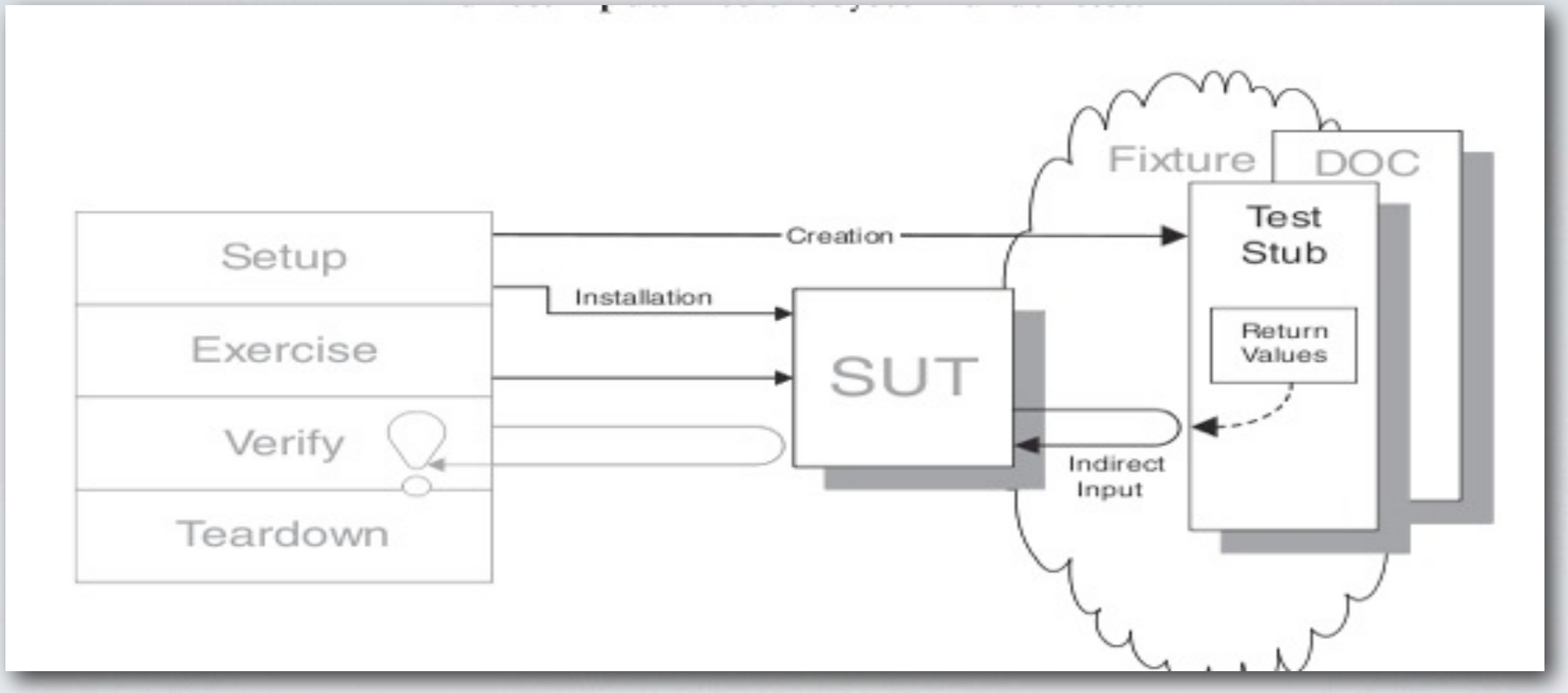
- Test Doubles, Stubs, Mocks, Fake Objects… how could we work it out ?

TEST DOUBLE PATTERN



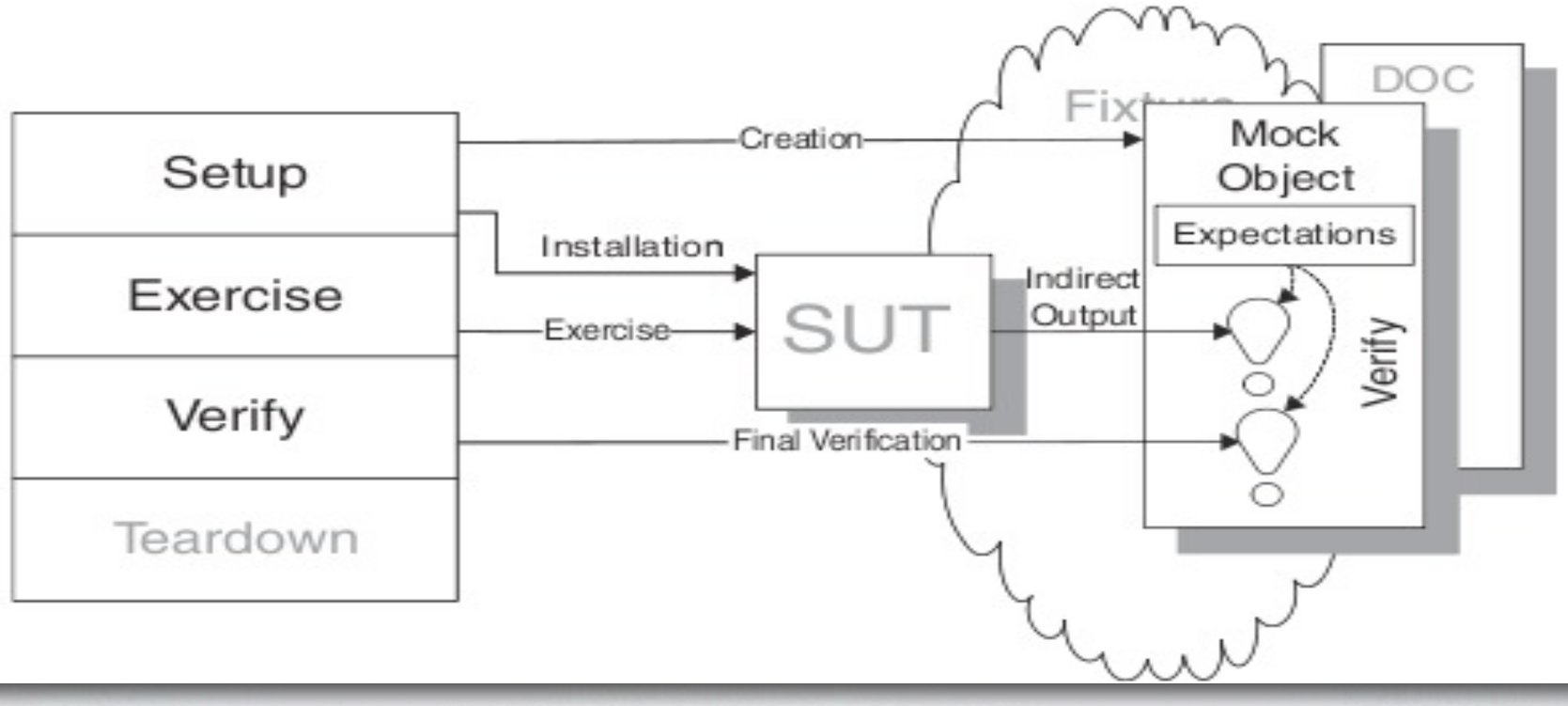
- **Q**: How can we verify logic independently when code it depends on is unusable?
  - **Q1**: How we can avoid slow tests ?
- **A**: We replace a component on which the SUT depends with a "test-specific equivalent."

TEST STUB PATTERN



- **Q**: How can we verify logic independently when it depends on indirect inputs from other software components ?
- **A**: We replace a real objects with a test-specific object that feeds the desired inputs into the SUT

MOCKS OBJECTS



- **Q**: How can we implement Behavior Verification for indirect outputs of the SUT ?
- **A**: We replace an object on which the SUT depends on with a test-specific object that verifies it is being used correctly by the SUT.

MOCK LIBRARIES

- Two main design philosophy:
  - **DSL** Libraries
  - **Record/Replay** Models Libraries

Record Replay Frameworks: First train mocks and then verify expectations

DSL Frameworks:

- Domain Specific Languages
- Specifications embedded in "Java" Code

MOCK LIBRARIES

- Two main design philosophy:
  - **DSL** Libraries
  - **Record/Replay** Models Libraries

Record Replay Frameworks: First train mocks and then verify expectations

DSL Frameworks:

- Domain Specific Languages
- Specifications embedded in "Java" Code

SOLUTION WITH JMOCK

```
import org.jmock.Expectations;
import org.jmock.integration.junit4.JUnitRuleMockery;
import org.junit.Before;
import org.junit.Test;

public class SchedulerTestWithJMock {

    private final JUnitRuleMockery context  = new JUnitRuleMockery();
    private final Queue queue = context.mock(Queue.class);
    private final Process process = context.mock(Process.class);

    private Scheduler s;

    @Before public void
    setUp(){
        this.s = new Scheduler();
    }

    @Test public void
    addProcessCallsMethodEnqueueOfQueue(){

        context.checking(new Expectations(){{
            oneOf(queue).enqueue(process);
        }});

        this.s.addQueue(queue);
        this.s.addProcess(process, queue);
    }
}
```

JMOCK MAIN FEATURES

JMOCK FEATURES (INTRO)

- JMock previous versions required subclassing
  - Not so smart in testing
  - Now directly integrated with Junit4
  - JMock tests requires more typing
- JMock API is extensible

JMOCK FEATURES

- JMock syntax relies heavily on chained method calls
  - Sometimes difficult to decipher and to debug

- **Common Patterns:**

```
invocation-count(mockobject).method(arguments);
inSequence(sequence-name);
when(state-machine.is(state-name));
will(action);
then(state-machine.is(new-state name));
```

1. TEST FIXTURE

```
import org.jmock.Expectations;
import org.jmock.integration.junit4.JUnitRuleMockery;
import org.junit.Before;
import org.junit.Test;

public class SchedulerTestWithJMock {

    private final JUnitRuleMockery context  = new JUnitRuleMockery();
```

- **Mockery** represents the **context**
- `JUnitRuleMockery` **replaces** the `@RunWith(JMock.class)` annotation
- JUnit4Mockery reports expectation failures as JUnit4 test failures

2. CREATE MOCK OBJECTS

```
private final Queue queue = context.mock(Queue.class);
private final Process process = context.mock(Process.class);
```

- References (fields and Vars) have to be **final**
  - Accessible from Anonymous Expectations

3. TESTS WITH EXPECTATIONS

```
context.checking(new Expectations(){{
    oneOf(queue).enqueue(process);
}});
```

- A test sets up it expectations in one or more **expectation** blocks
  - An expectation block can contain any number of expectations
  - Expectation blocks can be **interleaved** with calls to the code under test.

3. TESTS WITH EXPECTATIONS

```
context.checking(new Expectations(){{
    oneOf(queue).enqueue(process);
}});
```

- **Expectations** have the following structure:

```
invocation-count(mockobject).method(arguments);
inSequence(sequence-name);
when(state-machine.is(state-name));
will(action);
then(state-machine.is(new-state name));
```

WHAT ARE THOSE DOUBLE BRACES?

```
context.checking(new Expectations(){{
    oneOf(queue).enqueue(process);
}});
```

- Anonymous subclass of Expectations
- Baroque structure to provide a **scope** for setting expectations
  - Collection of expectation components
  - Is an example of **Builder Pattern**
  - Improves code completion

COOKBOOK: EXPECT A SEQUENCE OF INVOCATIONS

Expect that a sequence of method calls has been executed in the right order

```
public interface DummySequenceInterface {
    void first();
    void second();
    void third();
}

public class SequenceLauncher {
    public void startSequence(DummySequenceInterface seq) {
        seq.first();
        seq.second();
        seq.third();
    }
}
```

@Marco Zeuli

EXPECT A SEQUENCE OF INVOCATIONS

```java
import org.jmock.Expectations;
import org.jmock.Sequence;
import org.jmock.auto.Auto;
import org.jmock.auto.Mock;
import org.jmock.integration.junit4.JUnitRuleMockery;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;

public class TestSequenceLauncher {

    @Rule
    public final JUnitRuleMockery context = new JUnitRuleMockery();

    @Mock DummySequenceInterface seqInt;
    @Auto Sequence seq;

    private SequenceLauncher launcher;

    @Before
    public void setUp() {
        launcher = new SequenceLauncher();
    }

    @Test //This test should pass
    public void sequenceIsPerformedInTheCorrectOrder() {
        context.checking(new Expectations(){{
          oneOf(seqInt).first(); inSequence(seq);
          oneOf(seqInt).second(); inSequence(seq);
          oneOf(seqInt).third(); inSequence(seq);
        }});

        launcher.startSequence(seqInt);
    }

    @Test //This test should NOT pass
    public void sequenceIsNOTPerformedInTheCorrectOrder() {
        context.checking(new Expectations(){{
          oneOf(seqInt).second(); inSequence(seq);
          oneOf(seqInt).first(); inSequence(seq);
          oneOf(seqInt).third(); inSequence(seq);
        }});

        launcher.startSequence(seqInt);
    }

}
```

@Marco Zeuli

EXERCISE III

Roman Calculator

A SIMPLE EXAMPLE: THE ROMAN CALCULATOR

> Everyone always uses the same one, which is a Roman Numerals, but I'm going to give it a little twist, which is that I'll try and use a *Roman Numeral calculator* - not a *Roman Numeral converter* [...]

Source: https://github.com/hjwp/tdd-roman-numeral-calculator

PYTHON vs JAVA

- Language for *geeks*
- Multi-paradigm
- **Strong** Typed
- **Dynamic** Typed

- Language for "serious" guys
- Object Oriented Language
- **Strong** Typed
- **Static** Typed

DUCK TYPING

```
def half (n):
    return n/2.0
```

- Walks like a duck?
- Quacks like a duck?
- **Yes, It's a duck!**

Q: What is the **type** of the variable **n**

IS THERE SOMEONE THAT (REALLY) USES PYTHON?

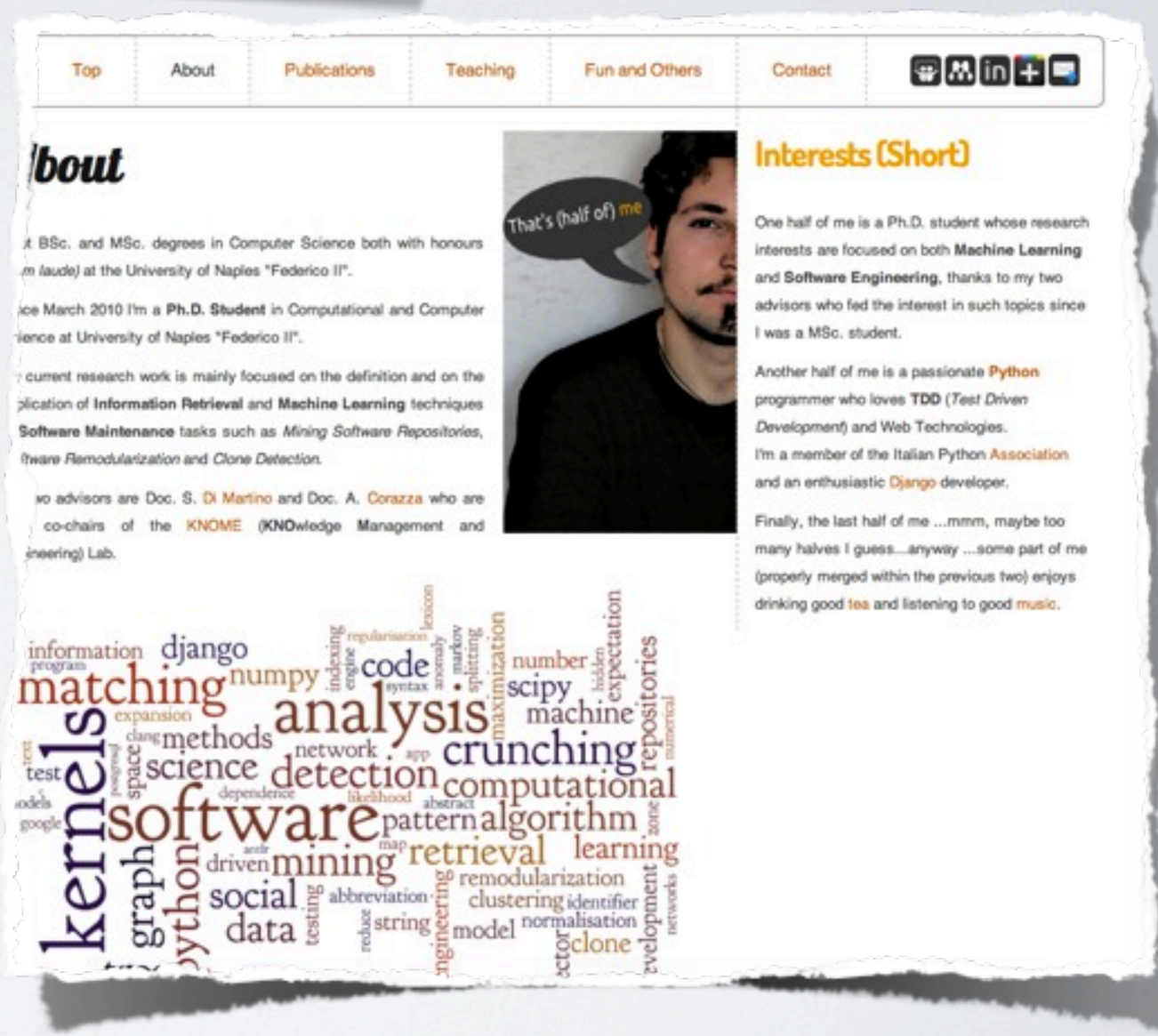
- IBM, **Google**, Microsoft, Sun, HP, **NASA**, Industrial Light and Magic
- **Google it!**
  - **site:microsoft.com** python
  - You'll get more than 9k hits

CONTACTS

http://wpage.unina.it/valerio.maggio

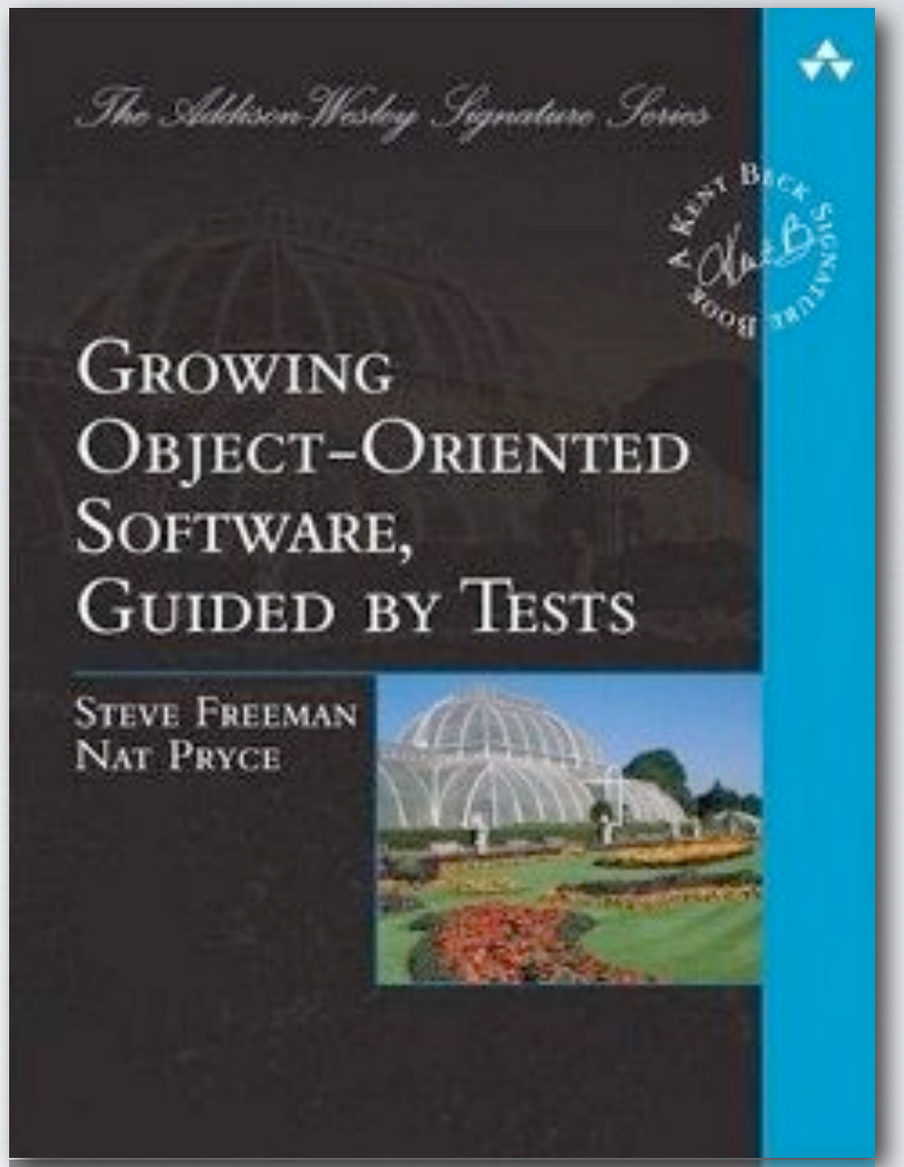
MAIL:

valerio.maggio@unina.it



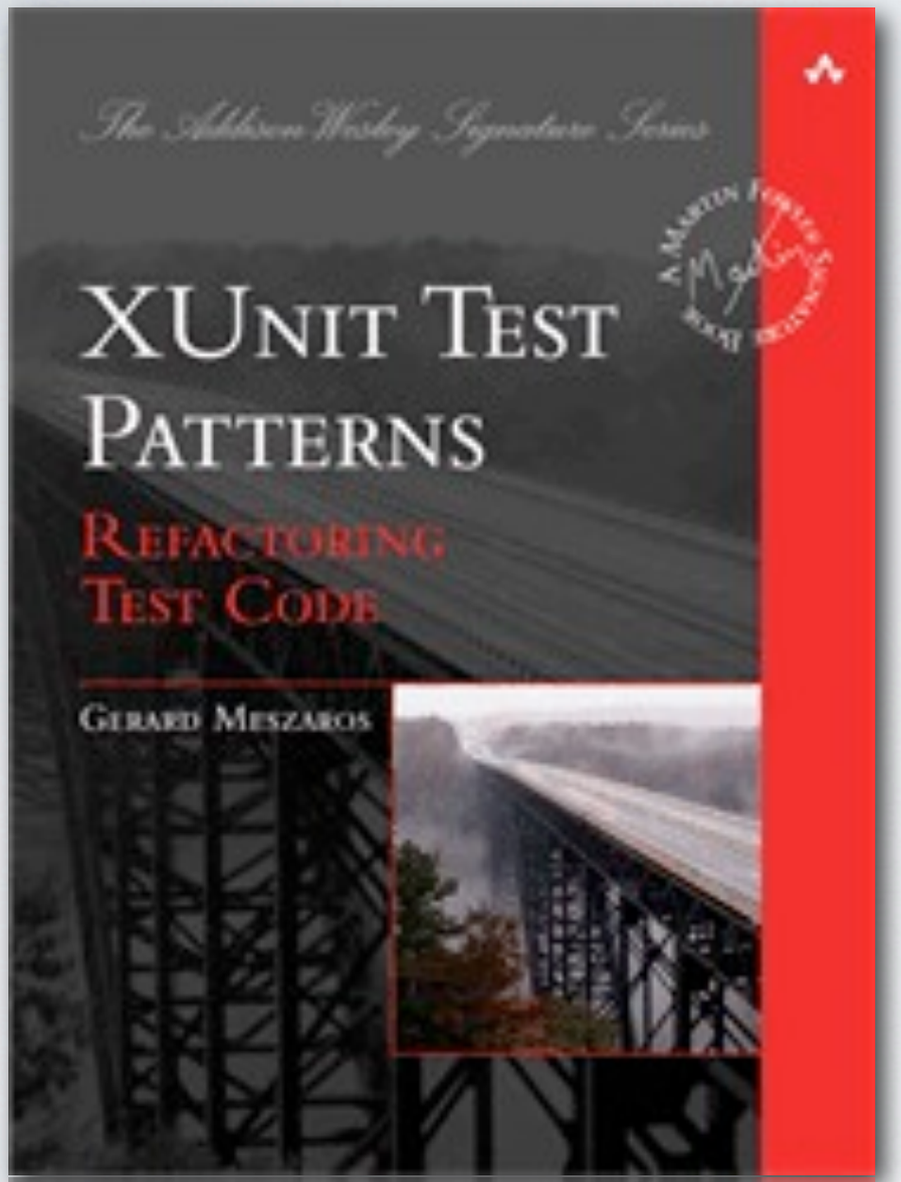
REFERENCES (1)

Growing Object-Oriented Software, Guided By Tests
Freeman and Pryce, Addison Wesley 2010



JMock Project WebSite
(http://jmock.org)

REFERENCES (II)



xUnit Test Patterns: Refactoring Test Code Meszaros G., Pearson Education 2007