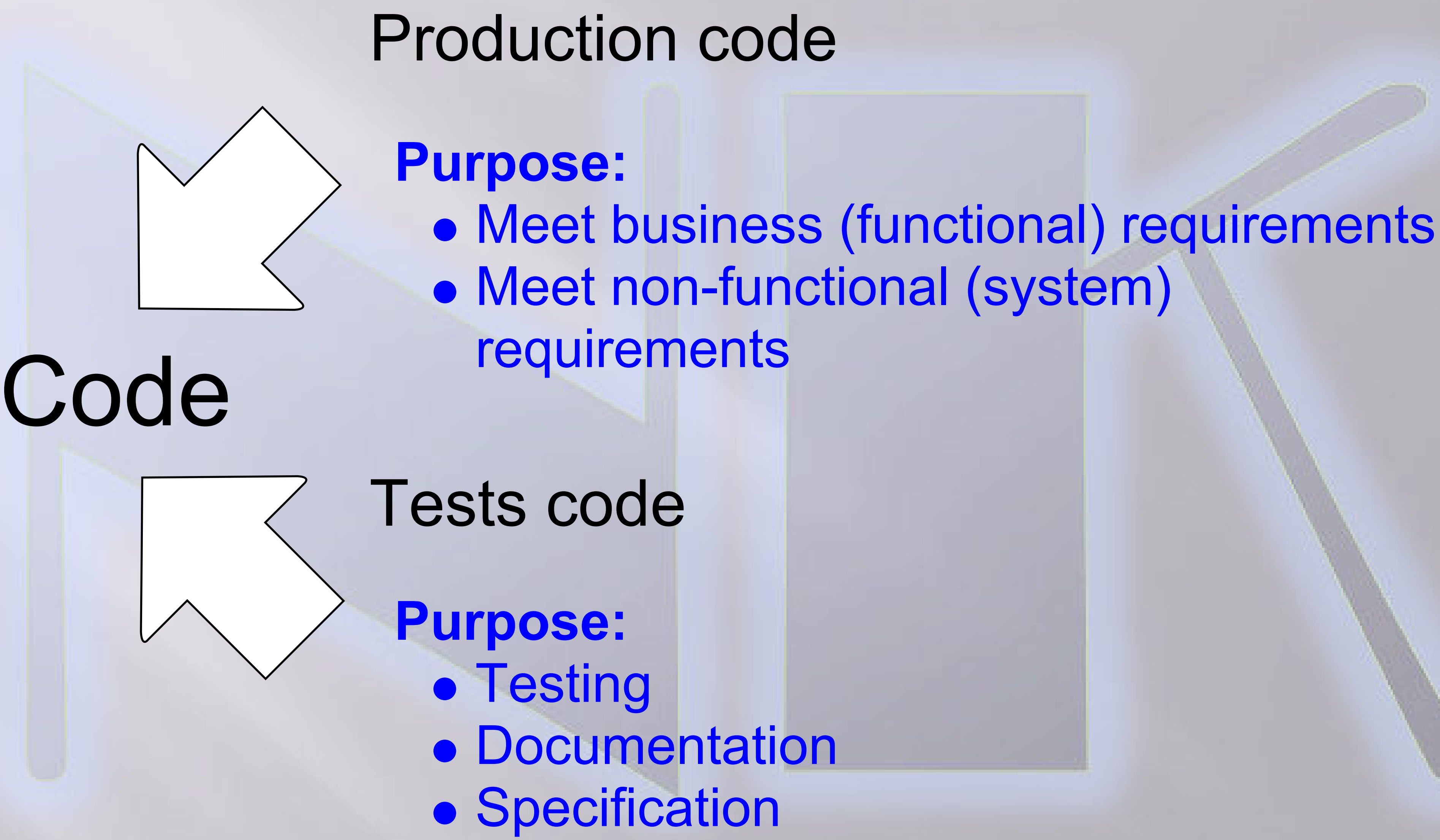
UNIT TESTING

best practices

www.nickokiss.com



Production code

Purpose:

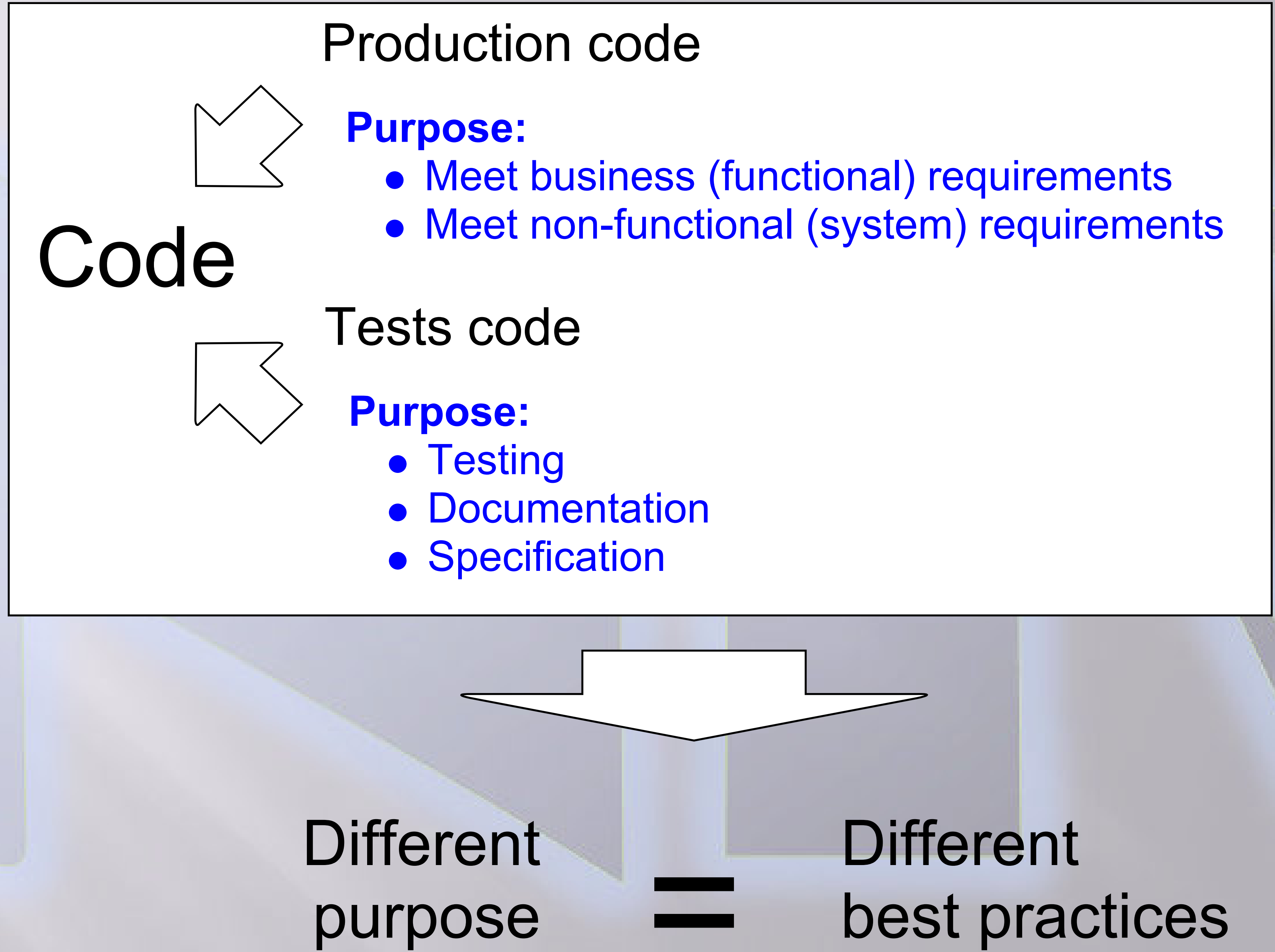
- Meet business (functional) requirements
- Meet non-functional (system) requirements

Code

Tests code

Purpose:

- Testing
- Documentation
- Specification



Code

Production code

Purpose:

- Meet business (functional) requirements
- Meet non-functional (system) requirements

Tests code

Purpose:

- Testing
- Documentation
- Specification

Different purpose = Different best practices

Unit testing best practices

- 3 steps
- Fast
- Consistent
- Atomic
- Single responsibility
- Tests isolation
- Environment isolation
- Classes isolation
- Fully automated
- Self-descriptive

- No conditional logic
- No loops
- No exception catching
- Assertions
- Informative assertion messages
- No test logic in production code
- Separation per business module
- Separation per type

Unit testing best practices

- **3 steps**
- Fast
- Consistent
- Atomic
- Single responsibility
- Tests isolation
- Environment isolation
- Classes isolation
- Fully automated
- Self-descriptive
- No conditional logic
- No loops
- No exception catching
- Assertions
- Informative assertion messages
- No test logic in production code
- Separation per business module
- Separation per type

3 steps

- Prepare an input
- Call a method
- Check an output

3 steps (5 steps)

- Set up
- Prepare an input
- Call a method
- Check an output
- Tear down

Unit testing best practices

- 3 steps
- **Fast**
- Consistent
- Atomic
- Single responsibility
- Tests isolation
- Environment isolation
- Classes isolation
- Fully automated
- Self-descriptive

- No conditional logic
- No loops
- No exception catching
- Assertions
- Informative assertion messages
- No test logic in production code
- Separation per business module
- Separation per type

Execution time - be fast

Why is it so important?

- Frequent execution
  - Several times per day *[Test-after development]*
  - Several times per hour *[Test driven development]*
  - Every few minutes *[IDE - Execute after save]*
- Execution in groups
  - 10 tests = Execution time x 10
  - 100 tests = Execution time x 100
  - Weak link: a slow test slows the whole suite

Execution time - be fast

What are the good numbers?

- Expected average execution time in unit testing
  - Single test: **<200ms**
  - Small suite: **<10s**
  - All tests suite: **<10min**

Unit testing best practices

- 3 steps
- Fast
- **Consistent**
- Atomic
- Single responsibility
- Tests isolation
- Environment isolation
- Classes isolation
- Fully automated
- Self-descriptive

- No conditional logic
- No loops
- No exception catching
- Assertions
- Informative assertion messages
- No test logic in production code
- Separation per business module
- Separation per type

Consistent

Multiple invocations of the test should consistently return true or consistently return false, provided no changes was made on code.

What code can cause problems?

- `Date currentDate = new Date();`
- `int value = random.nextInt(100);`

How to deal with this?

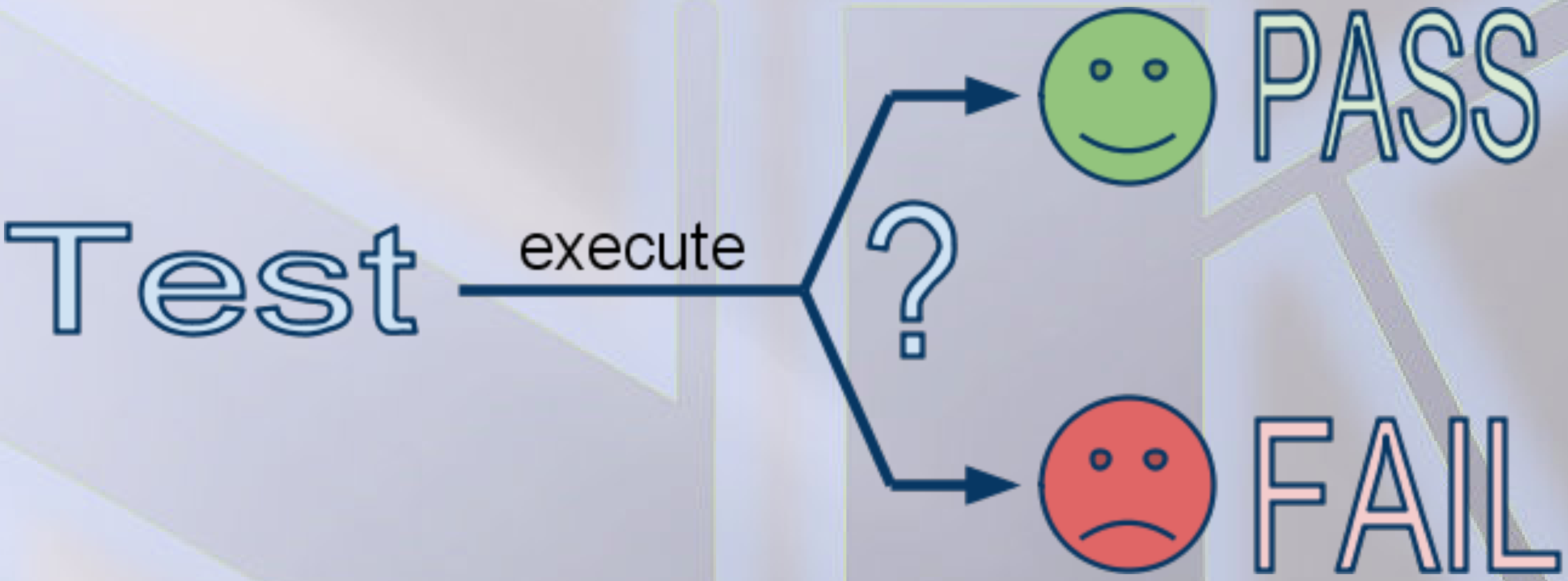
- Mocks
- Dependency injection

Unit testing best practices

- 3 steps
- Fast
- Consistent
- **Atomic**
- Single responsibility
- Tests isolation
- Environment isolation
- Classes isolation
- Fully automated
- Self-descriptive

- No conditional logic
- No loops
- No exception catching
- Assertions
- Informative assertion messages
- No test logic in production code
- Separation per business module
- Separation per type

Atomic



- Only two possible results: Pass or Fail
- No partially successful tests
- A test fails -> The whole suite fails
- Broken window effect

Unit testing best practices

- 3 steps
- Fast
- Consistent
- Atomic
- **Single responsibility**
- Tests isolation
- Environment isolation
- Classes isolation
- Fully automated
- Self-descriptive

- No conditional logic
- No loops
- No exception catching
- Assertions
- Informative assertion messages
- No test logic in production code
- Separation per business module
- Separation per type

Single responsibility

One test should be responsible for one scenario only.

Test behaviour, not methods:

- One method, **multiple** behaviours ⇨ **Multiple** tests
- **One** behaviour, multiple methods ⇨ **One** test
  - A method calls private and protected methods
  - A method calls very simple public methods
    (Especially: getters, setters, value objects, simple constructors)
- Multiple asserts in the same test - acceptable as long as they check the same behaviour

Single responsibility

- One method, multiple behaviours

☹

```
testMethod(){
    ...
    assertTrue(behaviour1);
    assertTrue(behaviour2);
    assertTrue(behaviour3);
}
```

☺

```
testMethodCheckBehaviour1(){
    ...
    assertTrue(behaviour1);
}
testMethodCheckBehaviour2(){
    ...
    assertTrue(behaviour2);
}
testMethodCheckBehaviour3(){
    ...
    assertTrue(behaviour3);
}
```

Single responsibility

- Behaviour1 = condition1 + condition2 + condition3
- Behaviour2 = condition4 + condition5

```
testMethodCheckBehaviours(){
    ...     assertTrue
(condition1);     assertTrue
(condition2);     assertTrue
(condition3);
    ...     assertTrue
(condition4);
assertTrue(condition5);
}
```

```
testMethodCheckBehaviour1(){
    ...
    assertTrue(condition1);
    assertTrue(condition2);
    assertTrue(condition3);
}
testMethodCheckBehaviour2(){
    ...
    assertTrue(condition4);
    assertTrue(condition5);
}
```

Unit testing best practices

- 3 steps
- Fast
- Consistent
- Atomic
- Single responsibility
- **Tests isolation**
- Environment isolation
- Classes isolation
- Fully automated
- Self-descriptive

- No conditional logic
- No loops
- No exception catching
- Assertions
- Informative assertion messages
- No test logic in production code
- Separation per business module
- Separation per type

Tests isolation

Tests should be independent from one another

- Different execution order - the same results
- No state sharing
- Instance variables
  - JUnit - separated
  - TestNG - shared

Unit testing best practices

- 3 steps
- Fast
- Consistent
- Atomic
- Single responsibility
- Tests isolation
- **Environment isolation**
- Classes isolation
- Fully automated
- Self-descriptive
- No conditional logic
- No loops
- No exception catching
- Assertions
- Informative assertion messages
- No test logic in production code
- Separation per business module
- Separation per type

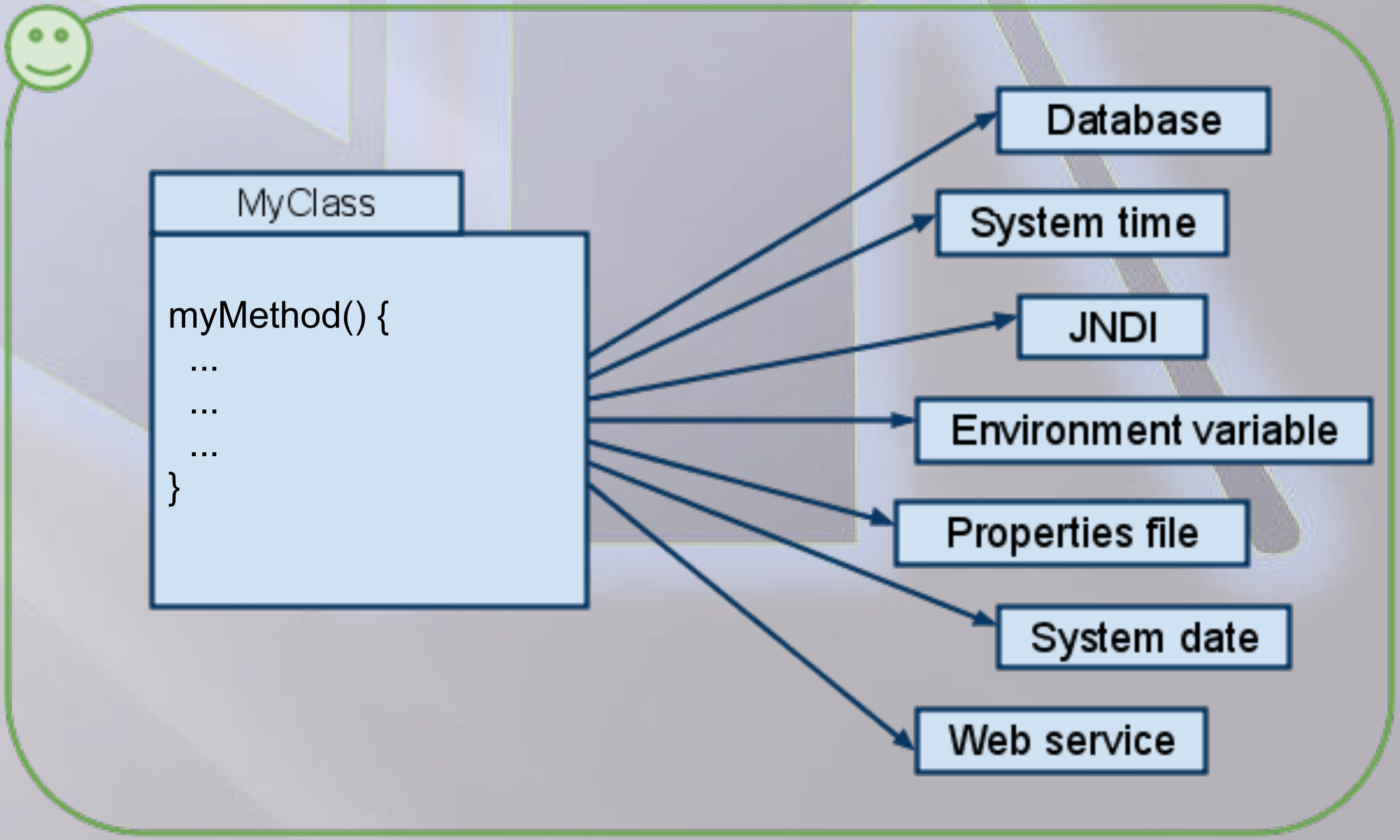
Environment isolation

Unit tests should be isolated from any environmental influences

- Database access
- Webservices calls
- JNDI look up
- Environment variables
- Property files
- System date and time

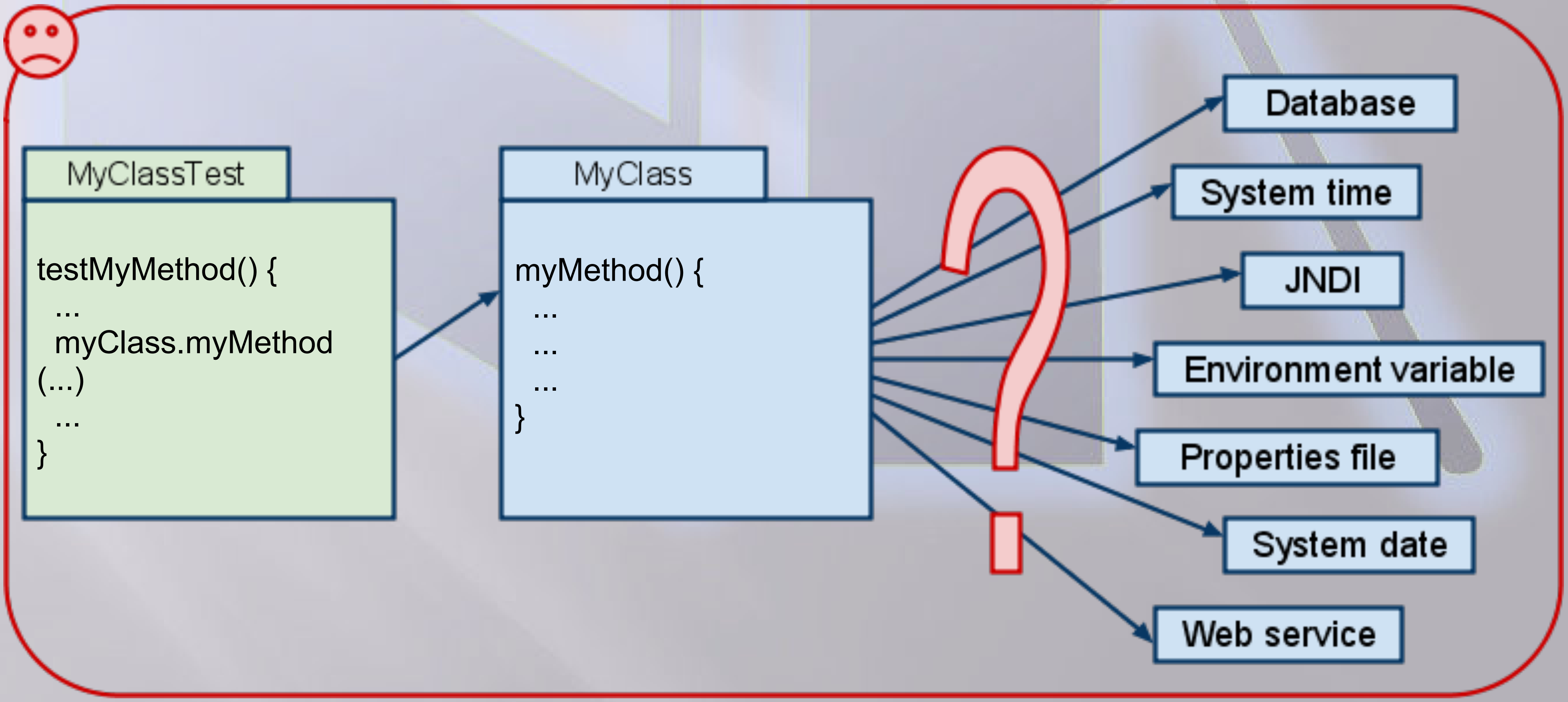
Environment isolation

Production code: A class heavily uses the environment



Environment isolation

Under unit testing: It doesn't work!



MyClassTest

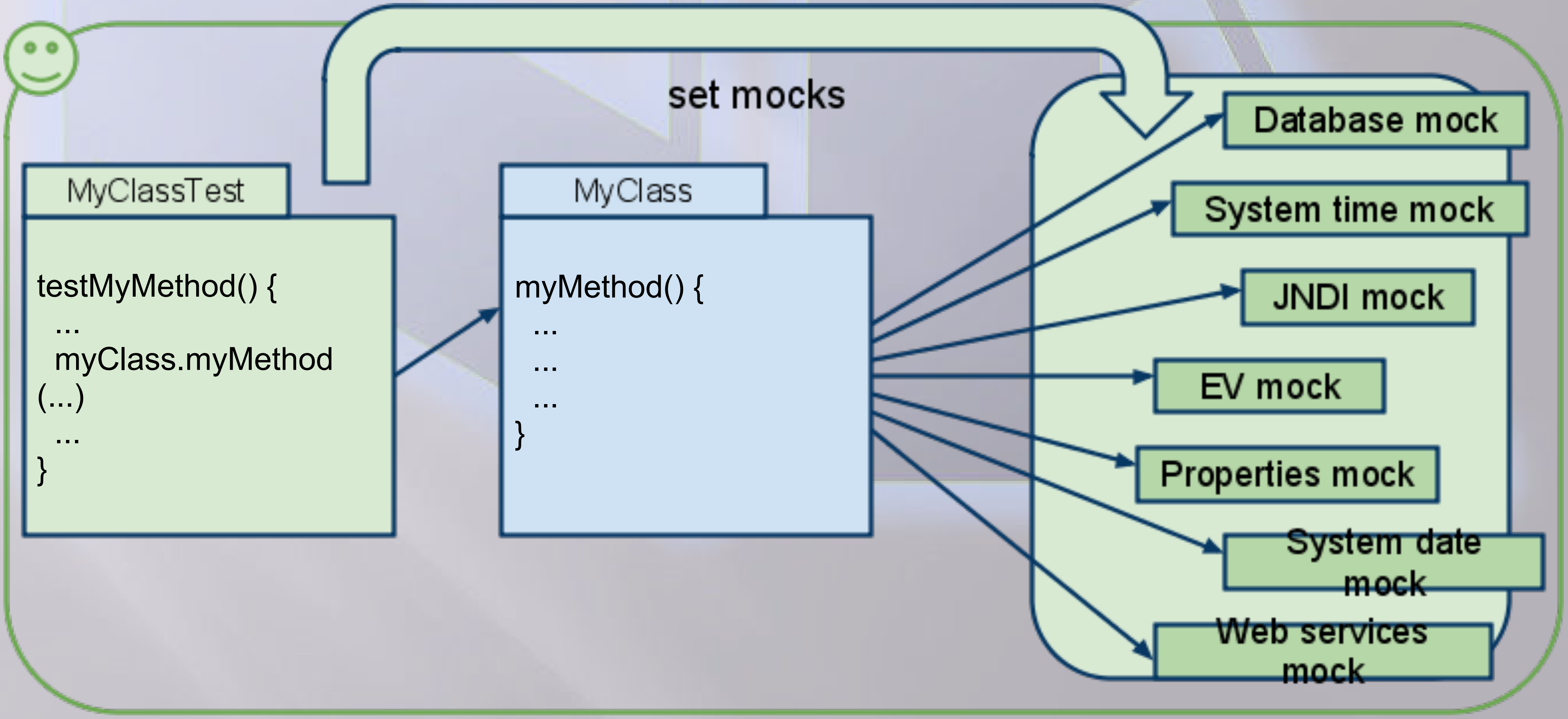
```
testMyMethod() {
  ...
  myClass.myMethod
(...)
  ...
}
```

MyClass

```
myMethod() {
  ...
  ...
  ...
}
```

Environment isolation

Solution: Use mocks

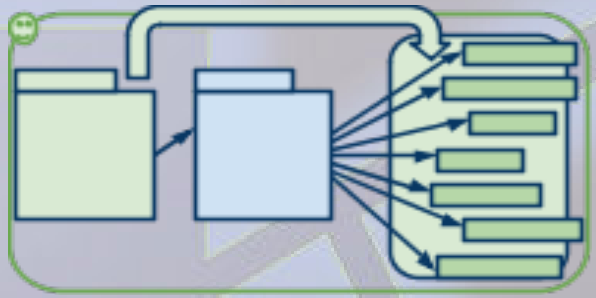


```
testMyMethod() {
  ...
  myClass.myMethod
(...)
  ...
}
```

```
myMethod() {
  ...
  ...
  ...
}
```

Environment isolation

Solution: Use mocks

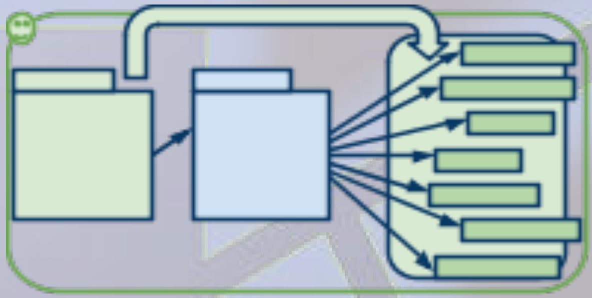


Advantages:

- No test logic in production code
- Fast
- Easy to write
- Easy to re-use

Environment isolation

Solution: Use mocks



Java mocking libraries (open source):

- EasyMock *[www.easymock.org]*
- JMock *[www.jmock.org]*
- Mockito *[www.mockito.org]*

Unit testing best practices

- 3 steps
- Fast
- Consistent
- Atomic
- Single responsibility
- Tests isolation
- Environment isolation
- **Classes isolation**
- Fully automated
- Self-descriptive

- No conditional logic
- No loops
- No exception catching
- Assertions
- Informative assertion messages
- No test logic in production code
- Separation per business module
- Separation per type

Classes isolation

The less methods are executed by the test, the better

(better code maintenance)

The less tests execute the method the better

(better tests maintenance)

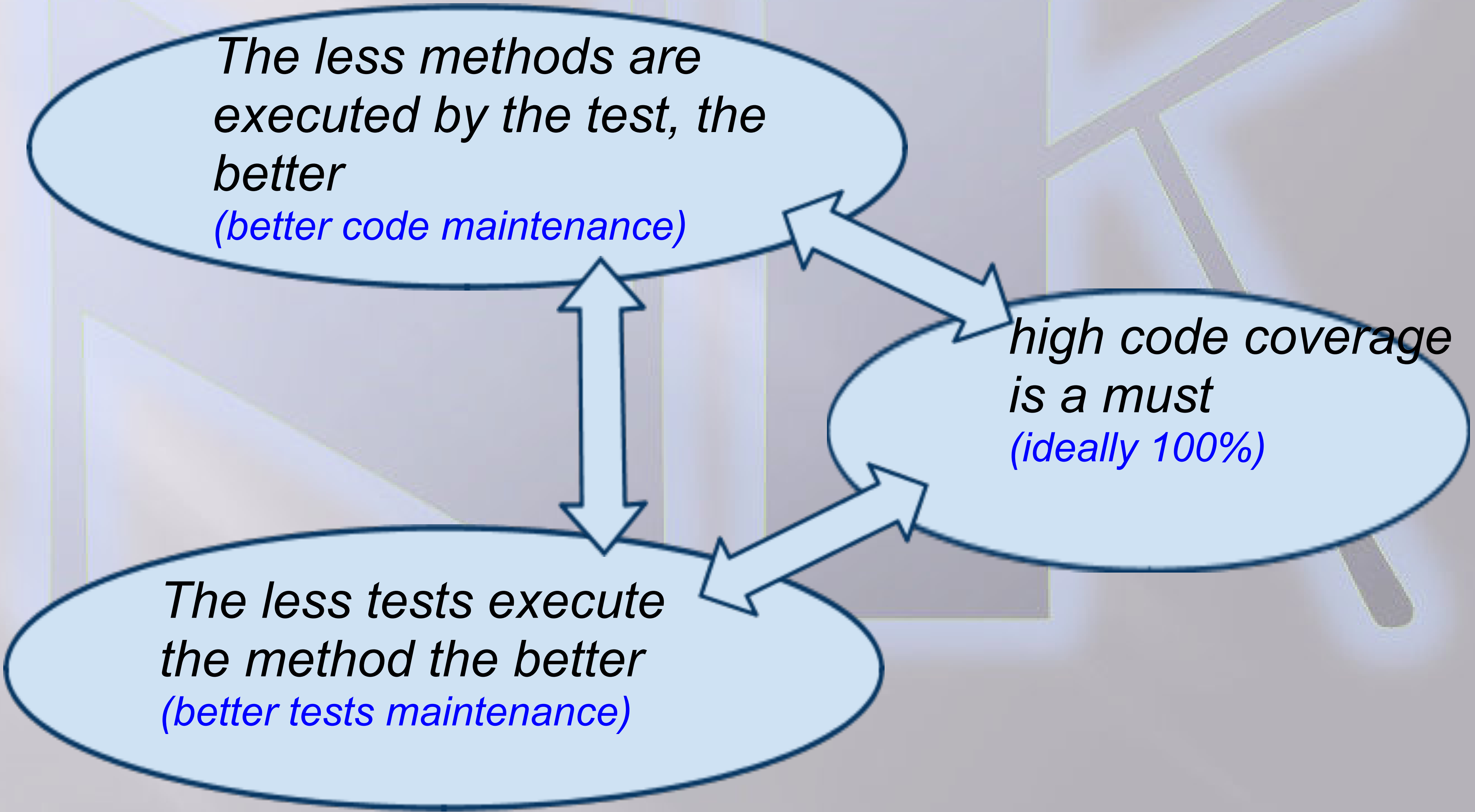
Classes isolation

BUT:

high code coverage is a must

(ideally 100%)

Classes isolation



The less methods are executed by the test, the better

(better code maintenance)

high code coverage is a must

(ideally 100%)

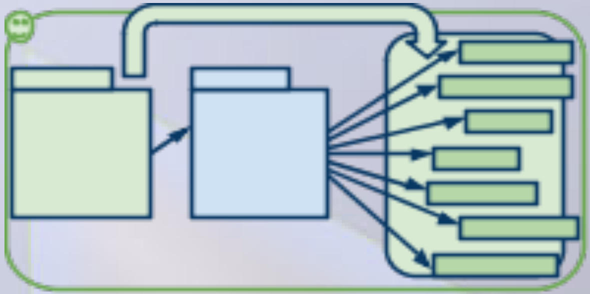
The less tests execute the method the better

(better tests maintenance)

Classes isolation

Let's come back to our previous example to ilustrate the problem

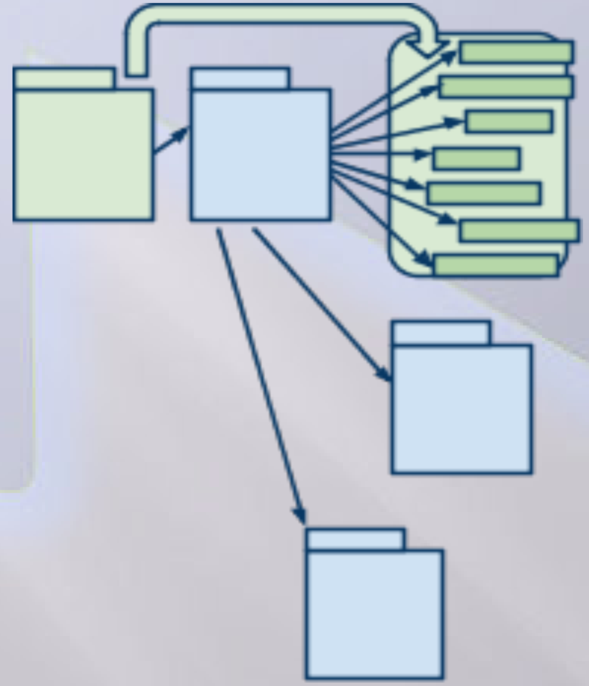
Step 0: The class under test has no dependencies



Classes isolation

Let's come back to our previous example to ilustrate the problem

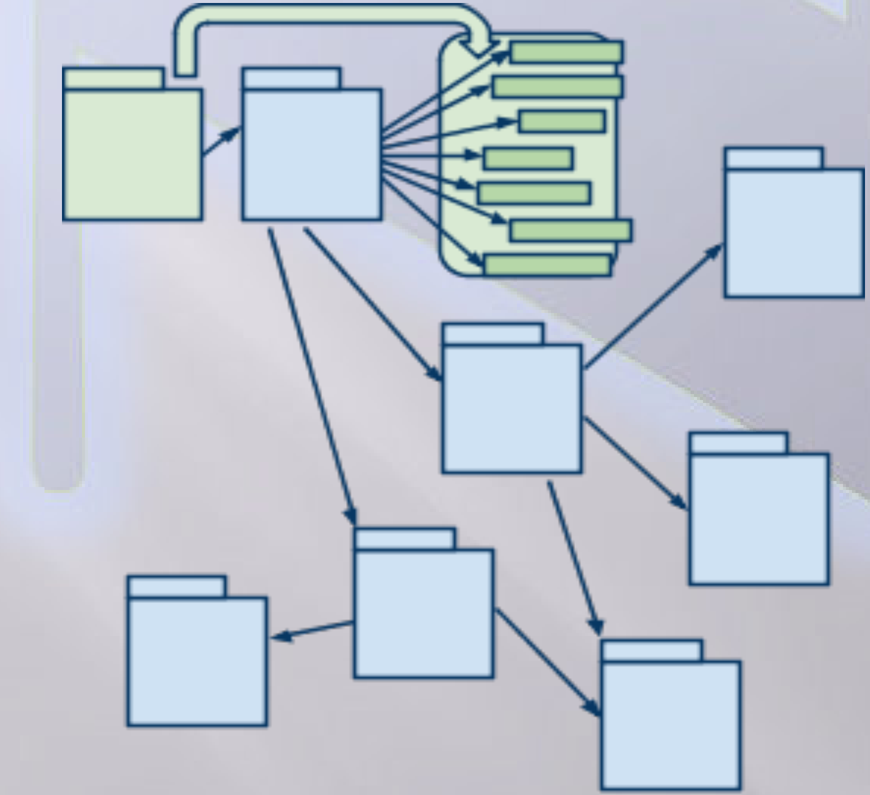
Step 1: The class depends on some other classes



Classes isolation

Let's come back to our previous example to ilustrate the problem

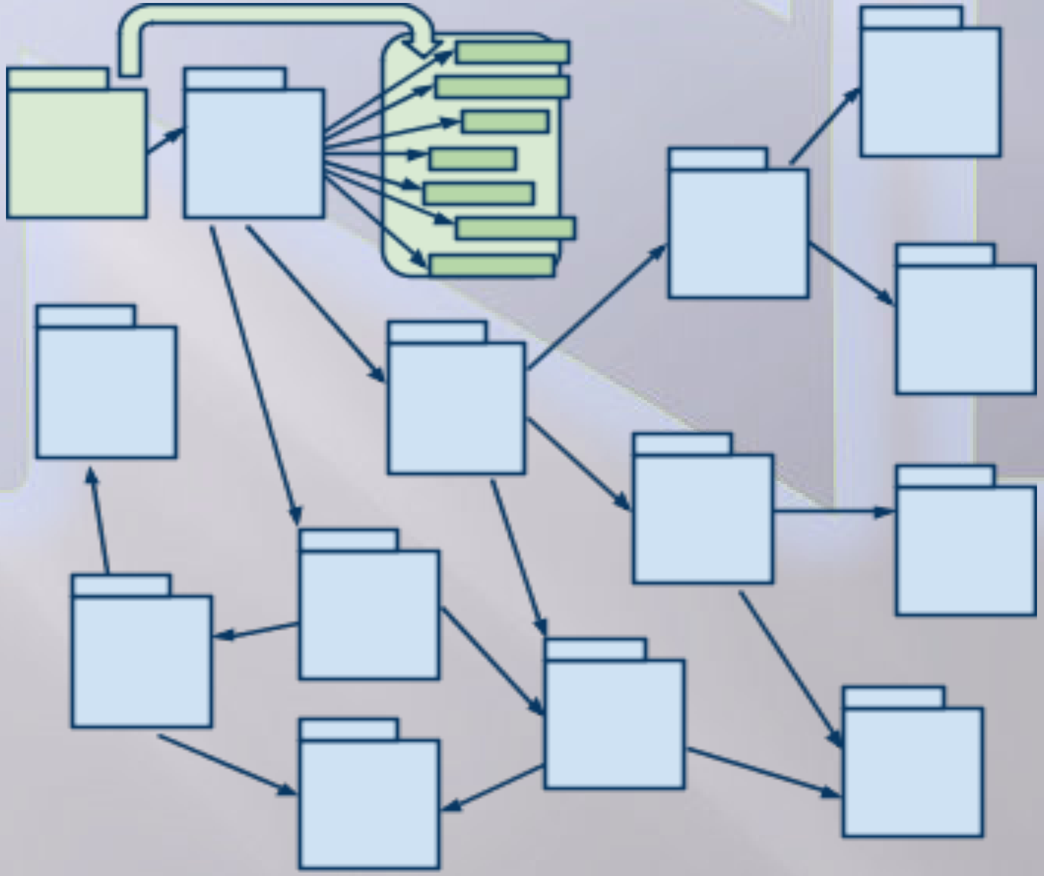
Step 2: Dependencies of dependencies



Classes isolation

Let's come back to our previous example to ilustrate the problem

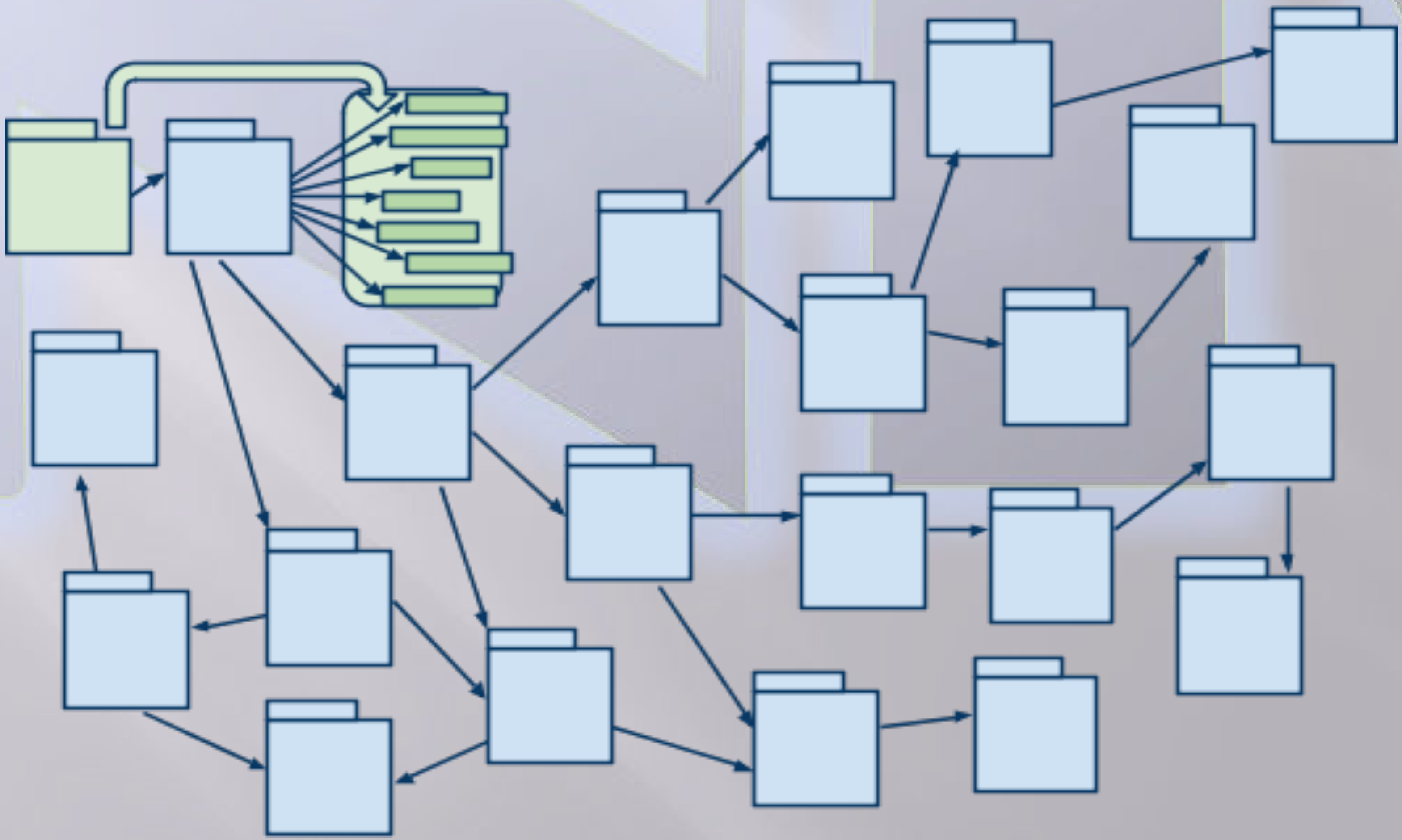
Step 3: Dependencies of dependencies of dependencies



Classes isolation

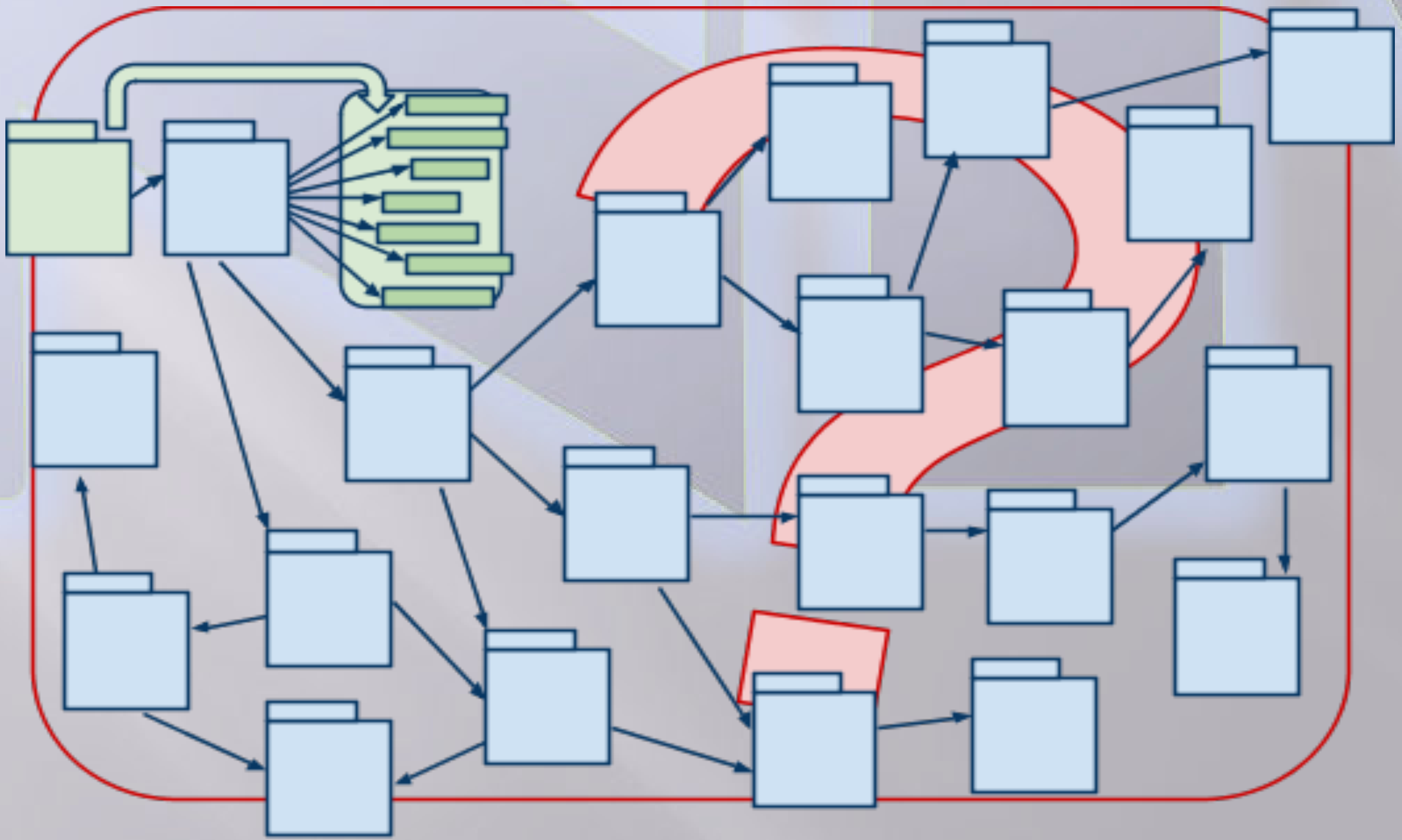
Let's come back to our previous example to ilustrate the problem

Step 4, 5, 6,...



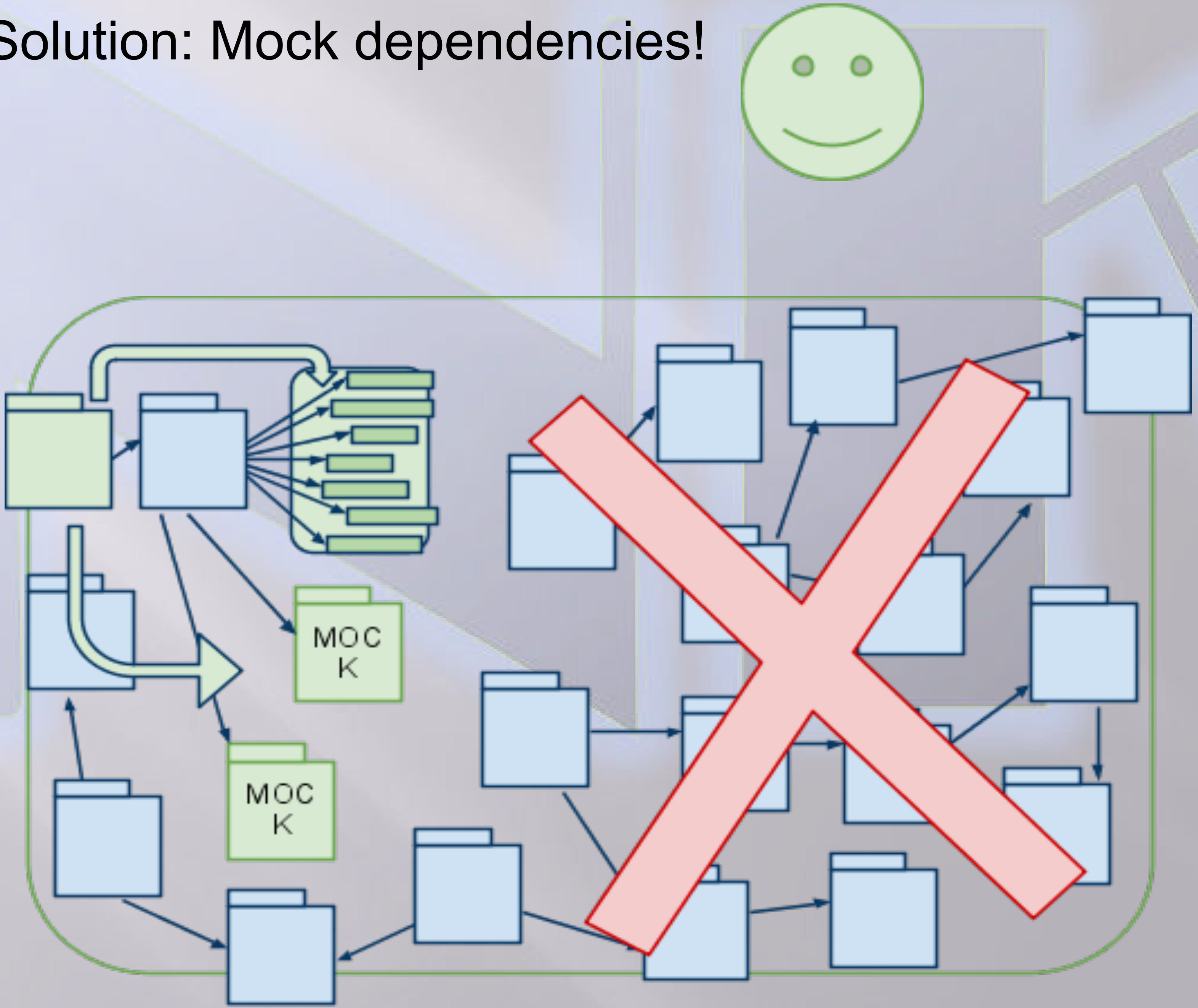
Classes isolation

Conclusion: Mocking the environment is not enough



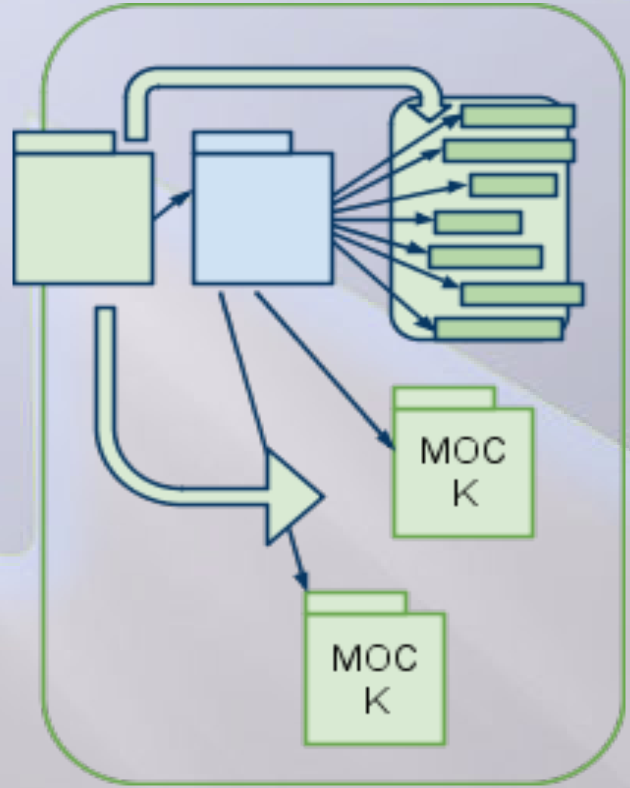
Classes isolation

Solution: Mock dependencies!



Classes isolation

Solution: Mock dependencies!



Classes isolation

Can be hard if code is not testable

How to write testable code?

- Don't call constructors inside a method. Use factories or dependency injection
- Use interfaces

Unit testing best practices

- 3 steps
- Fast
- Consistent
- Atomic
- Single responsibility
- Tests isolation
- Environment isolation
- Classes isolation
- **Fully automated**
- Self-descriptive

- No conditional logic
- No loops
- No exception catching
- Assertions
- Informative assertion messages
- No test logic in production code
- Separation per business module
- Separation per type

Fully automated

No manual steps involved into testing.

- Automated tests execution
- Automated results gathering
- Automated decision making (success or failure)
- Automated results distribution
    - Email
    - IM
    - System tray icon
    - Dashboard web page
    - IDE integration
    - Lava lamps

Unit testing best practices

- 3 steps
- Fast
- Consistent
- Atomic
- Single responsibility
- Tests isolation
- Environment isolation
- Classes isolation
- Fully automated
- **Self-descriptive**

- No conditional logic
- No loops
- No exception catching
- Assertions
- Informative assertion messages
- No test logic in production code
- Separation per business module
- Separation per type

Self-descriptive

Unit test = development level documentation

Unit test = method specification
which is always up to date

Self-descriptive

Unit test must be easy to read and understand

- Variable names
- Method names } Self-descriptive
- Class names
- No conditional logic
- No loops

Unit testing best practices

- 3 steps
- Fast
- Consistent
- Atomic
- Single responsibility
- Tests isolation
- Environment isolation
- Classes isolation
- Fully automated
- Self-descriptive

- **No conditional logic**
- No loops
- No exception catching
- Assertions
- Informative assertion messages
- No test logic in production code
- Separation per business module
- Separation per type

No conditional logic

Correctly written test contains no "if" or "switch" statements.

No uncertainty

- All input values should be known
- Method behaviour should be predictible
- Expected output should be strict

Split the test into two (or more) tests instead of adding "if" or "switch" statement.

No conditional logic

- One test, multiple conditions

☹

```
testMethodBeforeOrAfter(){
    ...
  if (before) {
   assertTrue(behaviour1);
  } else if (after) {
   assertTrue(behaviour2);
  } else { //now
   assertTrue(behaviour3);
  }
}
```

☺

```
testMethodBefore(){
    before = true;
    assertTrue(behaviour1);
}
testMethodAfter(){
    after = true;
    assertTrue(behaviour2);
}
testMethodNow(){
    before = false;
    after = false;
    assertTrue(behaviour3);
}
```

Unit testing best practices

- 3 steps
- Fast
- Consistent
- Atomic
- Single responsibility
- Tests isolation
- Environment isolation
- Classes isolation
- Fully automated
- Self-descriptive

- No conditional logic
- **No loops**
- No exception catching
- Assertions
- Informative assertion messages
- No test logic in production code
- Separation per business module
- Separation per type

No loops

High quality test contains no "while", "do-while" or "for" statements.

Typical scenarios involving loops:

- Hundreds of repetitions
- A few repetitions
- Unknown number of repetitions

No loops

Case 1: Hundreds of repetitions

If some logic in a test has to be repeated hundreds of times, it probably means that the test is too complicated and should be simplified.

No loops

Case 2: A few repetitions

Repeating things several times is OK, but then it's better to type the code explicitly without loops.You can extract the code which needs to be repeated into method and invoke it a few times in a row.

No loops

Case 3: Unknown number of repetitions

If you don't know how many times you want to repeat the code and it makes you difficult to avoid loops, it's very likely that your test is incorrect and you should rather focus on specifying more strict input data.

Unit testing best practices

- 3 steps
- Fast
- Consistent
- Atomic
- Single responsibility
- Tests isolation
- Environment isolation
- Classes isolation
- Fully automated
- Self-descriptive

- No conditional logic
- No loops
- **No exception catching**
- Assertions
- Informative assertion messages
- No test logic in production code
- Separation per business module
- Separation per type

No exception catching

- *Catch an exception only if it's expected*
- *Catch only expected type of an exception*
- *Catch expected exception and call "fail" method*
- *Let other exceptions go uncatched*

Catching expected exception

```
testThrowingMyException(){
    try {
      myMethod(param);
      fail("MyException expected");
    } catch(MyException ex) {
      //OK
    }
}
```

Unit testing best practices

- 3 steps
- Fast
- Consistent
- Atomic
- Single responsibility
- Tests isolation
- Environment isolation
- Classes isolation
- Fully automated
- Self-descriptive

- No conditional logic
- No loops
- No exception catching
- **Assertions**
- Informative assertion messages
- No test logic in production code
- Separation per business module
- Separation per type

Assertions

- *Use various types of assertions provided by a testing framework*
- *Create your own assertions to check more complicated, repetitive conditions*
- *Reuse your assertion methods*
- *Loops inside assertions can be a good practice*

Unit testing best practices

- 3 steps
- Fast
- Consistent
- Atomic
- Single responsibility
- Tests isolation
- Environment isolation
- Classes isolation
- Fully automated
- Self-descriptive

- No conditional logic
- No loops
- No exception catching
- Assertions
- **Informative assertion messages**
- No test logic in production code
- Separation per business module
- Separation per type

Informative assertion messages

By reading an assertion message only, one should be able to recognize the problem.

It's a good practice to include business logic information into assertion message.

Assertion messages:

- Improve documentation of the code
- Inform about the problem in case of test failure

Unit testing best practices

- 3 steps
- Fast
- Consistent
- Atomic
- Single responsibility
- Tests isolation
- Environment isolation
- Classes isolation
- Fully automated
- Self-descriptive

- No conditional logic
- No loops
- No exception catching
- Assertions
- Informative assertion messages
- **No test logic in production code**
- Separation per business module
- Separation per type

No test logic in production code

- *Separate unit tests and production code*
- *Don't create methods/fields used only by unit tests*
- *Use "Dependency Injection"*

Unit testing best practices

- 3 steps
- Fast
- Consistent
- Atomic
- Single responsibility
- Tests isolation
- Environment isolation
- Classes isolation
- Fully automated
- Self-descriptive

- No conditional logic
- No loops
- No exception catching
- Assertions
- Informative assertion messages
- No test logic in production code
- **Separation per business module**
- Separation per type

Separation per business module

- *Create suites of tests per business module*
- *Use hierarchical approach*
- *Decrease the execution time of suites by splitting them into smaller ones (per sub-module)*
- *Small suites can be executed more frequently*

Unit testing best practices

- 3 steps
- Fast
- Consistent
- Atomic
- Single responsibility
- Tests isolation
- Environment isolation
- Classes isolation
- Fully automated
- Self-descriptive

- No conditional logic
- No loops
- No exception catching
- Assertions
- Informative assertion messages
- No test logic in production code
- Separation per business module
- **Separation per type**

Separation per type

- Keep unit tests separated from integration tests
  - *Different purpose of execution*
  - *Different frequency of execution*
  - *Different time of execution*
  - *Different action in case of failure*

Thank you!

Find out more on:

http://www.nickokiss.com/2009/09/unit-testing.html