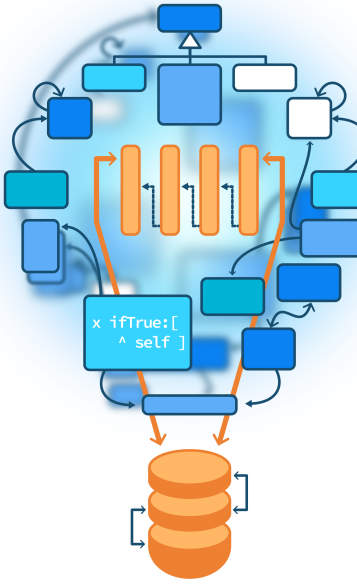
Test 101

The minimum you should know

S.Ducasse, L. Fabresse, G. Polito, and P. Tesone



Goal of the lecture

- How can you **trust** that a change did not destroy something?
- What is **my confidence** in the system?
- What is unit testing?
- How do I write tests?

Test main points

- When there is a change
  - Tests verify that what worked before still works
  - Tests are your **life insurance**: you get aware of a side effect and **regression**
- Tests are enablers of future evolution
- Tests reduce the **fear of change**
- Per se tests do not prevent bugs to happen but they reduce **unnoticed** bugs or side effects

About automation

A unit test that is not **automated does NOT EXIST!**

- Seriously!
- Repetition
- No human intervention

Unit tests

- Unit tests ensure that you get the specified behavior of a class
- Normally *unit* tests do test a single feature
- A test: one scenario, one point!

Anatomy of a test

A test:

- Creates a **context**
- Performs a **stimulus**: an action in the context
- **Checks** the result with **assertions**

Example: Testing duplicate set insertion

A test:

- Creates a context: Create an empty set
- Performs a stimulus: Add twice the same element
- Checks the results: Check that the set contains only one element

Set testcase in Pharo

```
TestCase subclass: #SetTest
   ...
```

```
SetTest >> testAdd
   | empty |
   "Context"
   empty := Set new.

   "Stimulus"
   empty add: 5.
   empty add: 5.

   "Check"
   self assert: empty size equals: 1.

SetTest run: #testAdd
```

Set testcase in Java (Junit40)

```
import java.util.Set;
import java.util.HashSet;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

class SetTest {
   @Test
   public void testAdd() {
      Set empty = new HashSet();    // context

      empty.add(5);                 //Stimulus
      empty.add(5);

      assertEquals(empty.size(),1); //Check
   }
}
```

Success, failures, and errors

- Success: a test passes
- A **failure** is a failed assertion, i.e., a verified property/assertion failed
- An **error** is an **unexpected** condition, i.e., an unexpected runtime error

A failure

If we get empty size returning 2 instead of 1.

```
SetTest >> testAdd
   | empty |
   empty := Set new.

   empty add: 5.
   empty add: 5.

   self assert: empty size equals: 1.
```

An error

Sending the message foobar: raises an exception.

```
SetTest >> testAdd
  | empty |
  empty := Set new.
  empty foobar: 5.
  self assert: empty size equals: 1.
```

How to reuse setting test context?

If a context is repeated among tests:

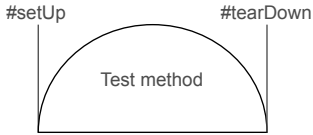
- duplication is never a good idea
- hampers future evolution

The framework offers the setUp method to create a context before any test execution.

setUp and tearDown messages

Executed **systematically** before and after each test run

- setUp allows us to specify and reuse the context
- tearDown to clean after test execution



Defining a setUp method

- Turn empty in an instance variable
- Just create a context, here empty is initialized to an empty set.

```
SetTestCase >> setUp
  empty := Set new
```

setUp is executed for you before any test execution

```
SetTestCase >> testAdd
  empty add: 5.
  empty add: 5.

  self assert: empty size equals: 1.
```

About writing tests

- Remember: Tests represent your trust in the system
- Build them incrementally
  - Do not need to focus on everything
  - When a new bug shows up, write a test
- Even better, write them before the code
  - Act as your first client, produce a better interface
- Active documentation is always in sync
- They have a cost: writing them, maintaining them. Make them worth
- But pay off is Huge

But I can’t cover everything!

Sure! Nobody can but:

- When someone discovers a defect in code, first write a test that demonstrates the defect.
- Then debug until the test succeeds.

Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead. Martin Fowler

Testing style: TDD

The style here is to write a few lines of code, then a test that should run, or even better, to write a test that won't run, then write the code that will make it run. Test Infected, Beck & Gamma, 1998

- Write unit tests that thoroughly test a single class
- Write tests as you develop (even before you implement your class!)
- Write tests for every new piece of functionality

(see next lecture)

Good tests

- Repeatable
- Do not require human intervention
- Are *self-described*
- Change less often than the system
- Tell a story

Conclusion

- Invest in tests
- Use Xtreme TDD: write a test, execute, debug, and code in the debugger (see following lecture)
- Tests are your best investment

Produced as part of the course on http://www.fun-mooc.fr

Advanced Object-Oriented Design and Development with Pharo

A course by
S.Ducasse, L. Fabresse, G. Polito, and P. Tesone

Inria LearningLab

2023