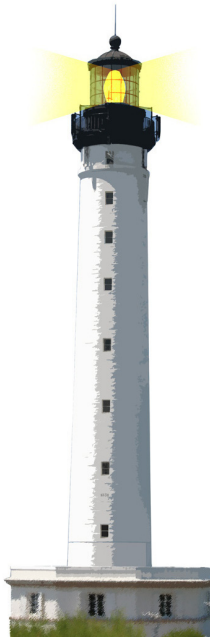
Test 101

The minimum you should know

S.Ducasse



Pharo

http://www.pharo.org

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 ** bugs or side effects

About automation

A test that is not ** does NOT exist!

- Repetition
- No human intervention

Unit tests

- Unit tests ensure that you get the specified behavior of a class
- Normally unit tests test a single class

Anatomy of a test

A test:

- Creates a context
- Performs a stimulus (an action on the context)
- Checks the results (with assertions)

Example: Testing set addition

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

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

```
SetTest >> testAdd
 | empty |
 empty := Set new.                "Context"
 empty add: 5.                    "Stimulus"
 empty add: 5.
 self assert: empty size equals: 1.      "Check"

SetTest run: #testAdd
```

Success, failures, and errors

- Success: a test passes
- A failure is a failed assertion, i.e., an anticipated problem that you test failed
- An error is a condition you didnt check for, i.e., a 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

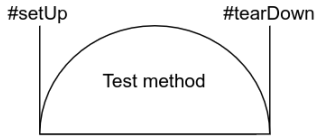
Sending the message add: raise an exception.

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

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



Defining a setUp method

Just create a context, here an empty set.

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

setUp is executed for you before any test execution

```
SetTestCase >> testOccurrences
 self
  assert: (empty occurrencesOf: 0)
  equals: 0.
 empty add: 5; add: 5.
 self
  assert: (empty occurrencesOf: 5)
  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, better interface
- Active documentation always in sync
- It has a cost: writing them, maintain them
- But pay off is Huge

But I cant 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."

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

'Developers should spend 25-50% of their time developing tests.'

Good tests

- Repeatable
- No human intervention
- "self-described"
- Change less often than the system
- Tells a story

Conclusion

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

A course by

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