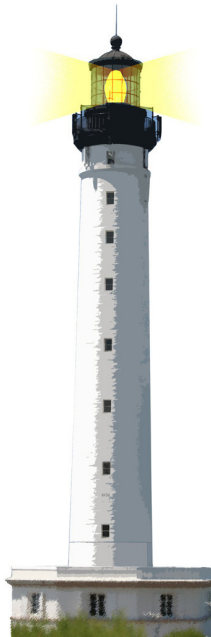
Tests

Why testing is Important?

S. Ducasse



Goal

- Why tests are important?
- What are their advantages?
- What are the techniques to write good tests?

Why testing?

- Tests are your life insurance
- Increase **trust** that a change did **not break** something
- Reduce the **fear** of changes
- Support code understanding
- Tests do not avoid breaking your system
- But **they show what you broke!**

Remember...

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

Automated tests are your life insurance

- Our brain is too small to **remember everything**
- Our brain focuses on our last action
- You write a test **once** and you execute it **million times**
- Programming is modeling the world and the world is **changing**

Automated tests ensure the software can evolve

- Tests make you **bold** in regards to changes
- Tests lower the fear of breaking
  - You can **try** and run the tests to get an idea
  - You can **explore** alternatives
  - You can understand that you **misunderstood** something

Test positive properties (1)

- **Find bugs** when they appear
- Improve customer trust
- Reproduce **complex** scenarii
- Guarantee old bugs are caught if reappear
- **Isolate** a problem

Some characteristics of a good test suite

- Check **extreme** cases (e.g., null, 0 and empty)
- Check complex cases (e.g., exceptions, network issues)
- 1 test for each bug (at least)
- Good **coverage**
- Check abstractions
- Check units independently

Understanding code: API and result

```
testConvert
  self assert: Color white convert equals: '#FFFFFF'.
  self assert: Color red convert equals: '#FF0000'.
  self assert: Color black convert equals: '#000000'
```

fromString: and convert interplay

```
testFromStringIsCoherentWithConvert
  | table aColorString |
  table := #('0' '1' '2' '3' '4' '5' '6' '7' '8' '9' 'A' 'B' 'C' 'D' 'E' 'F').

  table do: [ :each |
    aColorString := '#', each, each, '0000'.
    self assert: ((Color fromString: aColorString) convert sameAs: aColorString)].
```

Understanding code

You do not have to know how numbers are implemented to understand that this bitShift: is working.

```
testBitShift
  self assert: (2r11 bitShift: 2) equals: 2r1100.
  self assert: (2r1011 bitShift: −2) equals: 2r10.
```

Understanding code

You do not have to know how numbers are implemented to understand that this bitShift: is working.

```
testShiftOneLeftThenRightGetsOne
  "Shift 1 bit left then right and test for 1"

  1 to: 100 do: [:i |
    self
      assert: ((1 bitShift: i) bitShift: i negated)
      equals: 1].
```

Understanding code ;/

```
Color >> convert
  | s |
  s := '#000000' copy.
  s at: 2 put: (Character digitValue: ((rgb bitShift: −6 − RedShift) bitAnd: 15)).
  s at: 3 put: (Character digitValue: ((rgb bitShift: −2 − RedShift) bitAnd: 15)).
  s at: 4 put: (Character digitValue: ((rgb bitShift: −6 − GreenShift) bitAnd: 15)).
  s at: 5 put: (Character digitValue: ((rgb bitShift: −2 − GreenShift) bitAnd: 15)).
  s at: 6 put: (Character digitValue: ((rgb bitShift: −6 − BlueShift) bitAnd: 15)).
  s at: 7 put: (Character digitValue: ((rgb bitShift: −2 − BlueShift) bitAnd: 15)).
  ^ s
```

Understanding test ;)

```
ColorTest >> testAsHexString
  | table aColorString |
  self assert: Color white asHexString equals: 'FFFFFF'.
  self assert: Color red asHexString equals: 'FF0000'.
  self assert: Color black asHexString equals: '000000'.
```

Limit dependency to elements not under test

Imagine that we want to test a transformation of a piece of code

- If we depend on the compiler to get the test input
- It may break when the transformation is wrong, but also each time the compiler changes something!

Better have a setup that is independent of the compiler

- Manually build the test input and store it in a test setup

Think about API even in the test setup

Positive and negative tests

Positive

- If I do the normal stuff,
- It passes!
- Example: You can log in with the correct credentials

Negative

- If I do not behave correctly,
- It breaks!
- Example: You must not be able to load with incorrect credentials
- Example: It should raise an exception if given 0

Test positive properties (2)

- Give simple and **reproducible** examples
- **Executable** snippets
- Illustrate the API
- Give up-to-date documentation
- Check the conformity of new code
- Offer a **first client** to new code
- Force a '**customizable**' design

Characteristics of a good test suite

- Deterministic
- Self-explained
- Simple/Unit/Short: with few assertions (not 10th not hundred)
- Change less frequently than the rest:
  - **Test the API** not the implementation
  - **Limit** dependency to other elements
- Good code coverage

Conclusion

- Tests are important
- In particular in dynamically-typed languages
- Help deliver complex projects

A course by

S. Ducasse, G. Polito, and Pablo Tesone