Tests

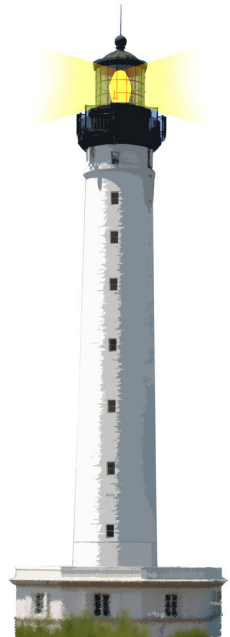
Why Testing is Important

Damien Cassou, Stéphane Ducasse and Luc Fabresse

WXSYY

Pharo

http://www.pharo.org



Goal

- why tests are important?
- what are their advantages?
- what are the techniques to write good tests?

Pros

- Finding problems
- Understanding code
- Increase trust
- Collateral pros

Finding Problems: Pros

- find bugs when they appear
- improve customer trust
- reproduce complex scenari
- check contracts of super types
- guarantee old bugs won’t come back
- isolate the problem

Finding Problems: Characteristics of a Good Test Suite

- check extreme cases (e.g., null, 0 and empty)
- check complex cases (e.g., exceptions, network pbs)
- 1 test for each bug (at least)
- good coverage
- check abstractions
- check units independently

Understanding Code

```
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 Code

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

Understanding Code

```
testConvert2
  | 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)].

  table do: [ :each |
    aColorString := '#', '00', each, each, '00'.
    self assert: ((Color fromString: aColorString) convert sameAs:
      aColorString)].

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

Understanding Code

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

testShiftOneLeftThenRight
  "Shift 1 bit left then right and test for 1"
  1 to: 100 do: [:i | self assert: ((1 bitShift: i) bitShift: i negated) =
      1].
```

Understanding Code: Pros

- give simple and reproducible examples
- explain an API
- give up-to-date documentation
- check conformity of new code
- offer a first client to new code
- force a modular design

Understanding Code: Characteristics of a Good Test Suite

- deterministic
- automatic
- self-explained
- simple
- unit

Increasing Trust: Pros

- accelerate bug detection
- accelerate new code checking
- ease refactorings
- prevent regressions

Increasing Trust: Characteristics of a Good Test Suite

- change less frequently than the rest
- good code coverage
- deterministic

Collateral Pros

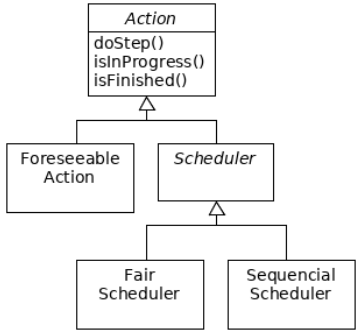
- improve feeling of customers who care
- allow for automatic bug fixing
- improve type inference
- provide examples to variable values

Testing Abstractions

How do you test contracts of abstract types?

Testing Abstractions

How do you test that one and only one state is active at any time?



Testing Abstractions

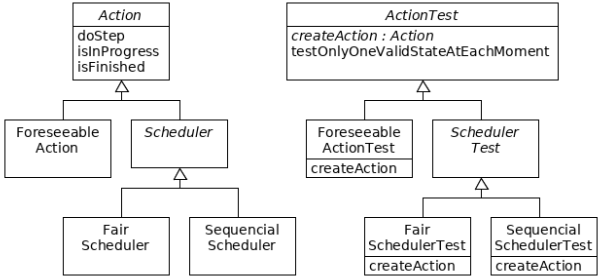

```
testOnlyOneValidStateAtEachMoment
  | action |
  action := self createAction.
  self assert: action isReady.
  self deny: action isInProgress. self deny: action isFinished.

  [ action isFinished ] whileFalse: [
    action doStep.
    self deny: action isReady.
    self assert: action isFinished = action isInProgress not ].

  self deny: action isReady. self deny: action isInProgress.
  self assert: action isFinished
```

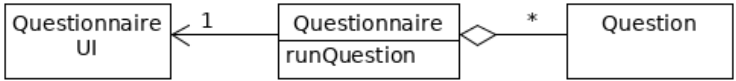
Testing Abstractions

- parallel hierarchies
- test must be in the highest abstraction
- factory method



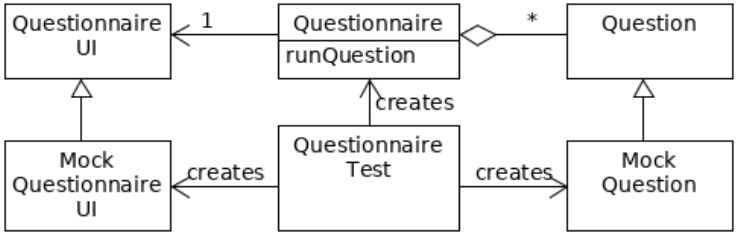
Mocking

How do you test that a questionnaire only accepts compatible answers from the user?

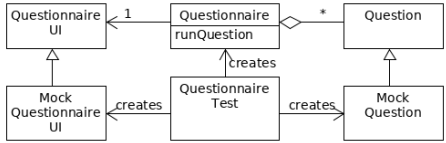


Mocking

How do you test that a questionnaire only accepts compatible answers from the user?

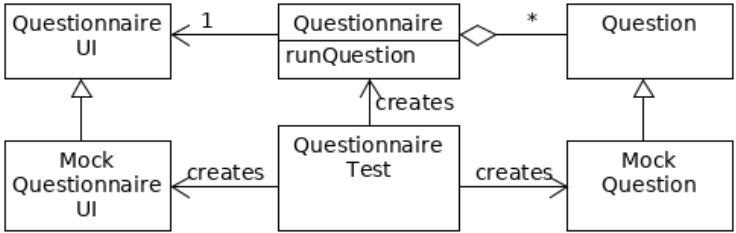


Mocking



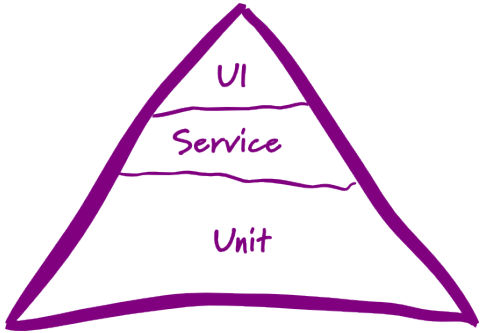

```
readAnswerAsLongAsItIsNotCompatible
  | nbRejectsBeforeAccept question ui |
  nbRejectsBeforeAccept := 3.
  question := MockQuestion new nbRejects:
      nbRejectsBeforeAccept.
  ui := MockQuestionnaireUI new.
  self assert: ui nbReadAnswers equals: 0.
  self assert: question nbAcceptAnswerCalls equals: 0.
  questionnaire runQuestion: question on: ui.
  self assert: ui nbReadAnswers equals: nbRejectsBeforeAccept
      + 1.
  self assert: question nbAcceptAnswerCalls equals:
```

Mocking



- mocks are reusable across tests
- mocks can be generated with mocking frameworks

Different Testing Levels



http://martinfowler.com/bliki/TestPyramid.html

A course by

and

in collaboration with