Cost-effective testing

Andrew P. Black

based on Chapter 9 of POODR

Reasons for testing

- document the interface
  - documents the non-interface methods too!
- design a good interface
  - shift from implementor to client
- finding bugs
- supports the abstractions
- expose design flaws
  - bad design => hard to test

What to test?

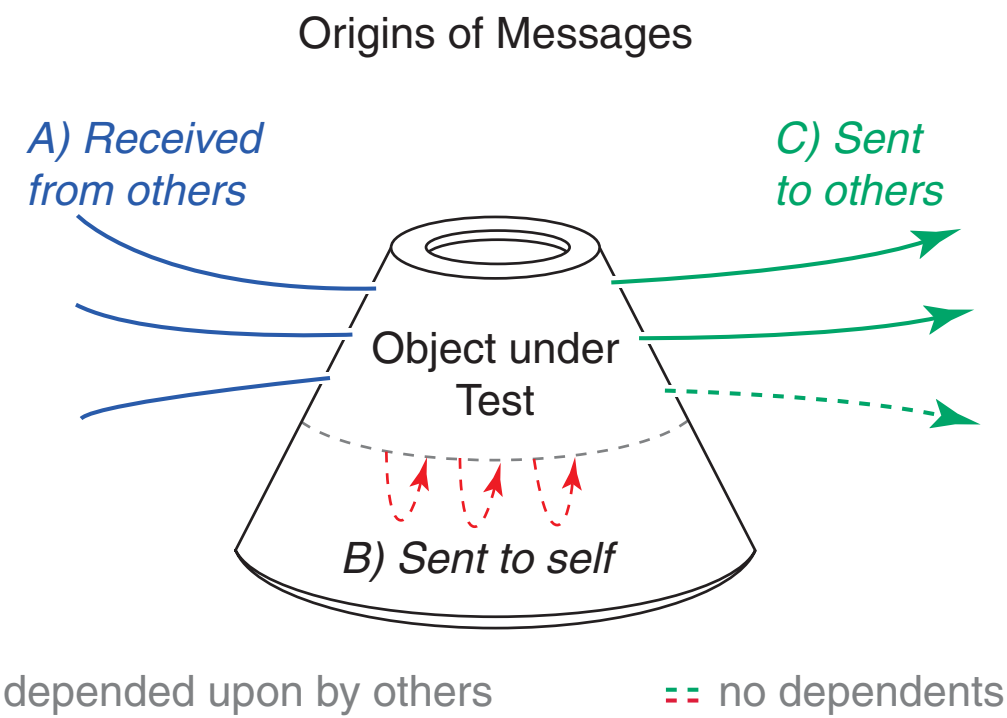


Figure 9.1 Objects under test are like space capsules, messages breach their boundaries.

What to test?

Black's Rule of Testing. For every test in the system, you should be able to identify some property for which the test increases your confidence. It's obvious that there should be no important property that you are not testing. This rule states the less obvious fact that there should be no test that does not add value to the system by increasing your confidence that a useful property holds. For example, several tests of the same property do no good. In fact, they do harm in two ways. First, they make it harder to infer the behaviour of the class by reading the tests. Second, because one bug in the code might then break many tests, they make it harder to estimate how many bugs remain in the code. So, have a property in mind when you write a test.

From: Pharo By Example, volume 1

Delete Unused Interfaces

Why?

Testing Private Methods

- The rules-of-thumb for testing private methods are:
  - Never write them, and
  - if you do, never ever test them,
  - unless of course it makes sense to do so.
- Therefore, be biased against writing these tests, but
  - do not fear to do so if this would improve your lot

How to test a private method

- Grace prohibits other objects from requesting confidential methods
- Java prohibits other objects from requesting private methods
  - unless the requesting object happens to have the same class as the target, which it won't if the requesting object is a testCase.
- What do you do?

Testing Outgoing Queries

- when the object under test makes a request of an observer method, *e.g.*,
  - `gear` requests `diameter` on `wheel`
- don't test it
  - but suppose that `wheel.diameter` answers incorrectly?

Testing Outgoing Commands

- when the object under test makes a request of a *command* method, *e.g.*,
  - `gear` requests `update` on `observer`
- demonstrate that the request is made
  - but don't make any assertions about the answer!
  - That's responsibility of `observer`'s tests

Mock Objects

- Mock objects test behaviour, rather than state:
  - rather than checking the result of a request,
  - they check that *the request was made*
- Mature languages will have a mocking framework
  - but you can also just code-up your own mock

A Mock Observer

Download mock.grace Help? Search Delete

```
1  class observer {
2
3      def log = list.empty
4      method update {
5          log.addLast "update"
6      }
7      method isCorrect {
8          (log.size == 1) && {log.first == "update"}
9      }
10 }
11
```

- Customized for this task

Using the observer

Download gear observer test.grace

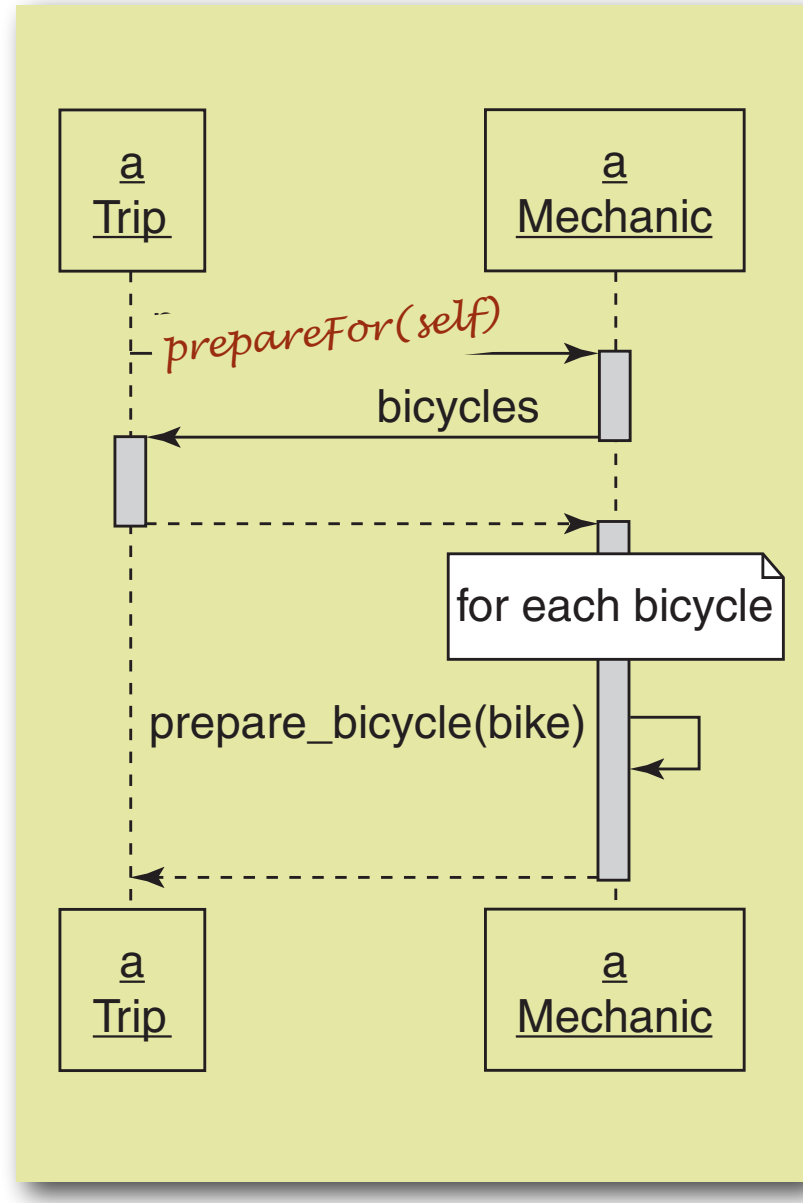
```
1  import "observable gear" as gAndW
2  import "gUnit" as gUnit
3  import "mock" as mock
4
5  def gearObserverTest = object {
6      class forMethod(m) {
7          inherit gUnit.testCaseNamed(m)
8
9          method testNotifiesObserverOfChainringChange {
10             def obs = mock.observer
11             def gear = gAndW.gearRing(42) cog (11) observer (obs)
12             gear.chainring := 52
13             assert(obs.isCorrect)
14         }
15         method testNotifiesObserverOfCogChange {
16             def obs = mock.observer
17             def gear = gAndW.gearRing(42) cog (11) observer (obs)
18             gear.cog := 17
19             assert(obs.isCorrect)
20         }
21     }
22 }
23
24 gUnit.testSuite.fromTestMethodsIn(gearObserverTest).runAndPrintResults
25
26
```

Why use a Mock?

- Why not just test that `observable gear` gets the correct answer from its update request to the observer?

Testing Interfaces

- Metz calls this "Testing Duck Types"
- Recall from Chapter 4:



Types as Interfaces

list of method names (with optional parameter & result types)

Note:

- you can define a type as an interface

```
type Preparer = interface {
    prepareFor(aTrip) → Done
    ...
}
```

- and use it to check your class:

```
class mechanic → Preparer {
    method prepareFor(aTrip) → Done { ... }
    ...
}
```

will raise a type error (at runtime) if the new object does not have the right methods

Or, use a Unit Test

Download mechanics are preparers test.grace

```
1  dialect "minitest"
2  import "preparers" as p
3
4  testSuite {
5      test "mechaninics are preparers" by {
6          assert (p.mechanic) hasType (p.Preparer)
7      }
8  }
9
10
```

- Include this with other tests of the `mechanic` class

Testing Inheritance Hierarchies

- Test that every object in the hierarchy has the right interface, and
- Test that concrete subclasses override all the required methods
  - in some languages, static checks may make these tests redundant
- Consider creating a *stub* subclass as a way of testing an abstract superclass
  - mock methods can check that the superclass makes *hook* requests

Summary

Tests are indispensable. Well-designed applications are highly abstract and under constant pressure to evolve; without tests these applications can neither be understood nor safely changed. The best tests are loosely coupled to the underlying code and test everything once and in the proper place. They add value without increasing costs.

A well-designed application with a carefully crafted test suite is a joy to behold and a pleasure to extend. It can adapt to every new circumstance and meet any unexpected need.