Testing with Concepts and Axioms

(in Magnolia)

Anya Helene Bagge

BLDL

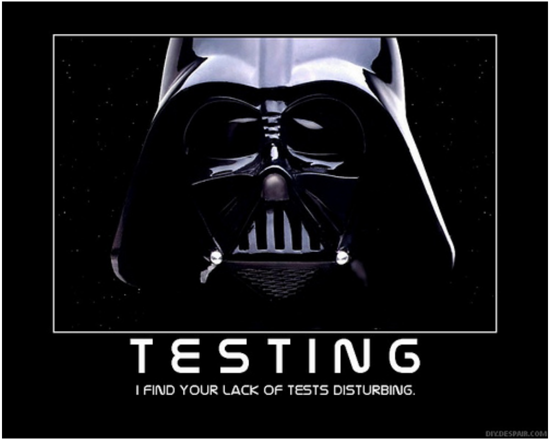
BLDL High Integrity Day
2014-02-11

Testing is good for you:

- Do it.
- A lot!

Unit testing:

- Test modules in isolation



Unit Testing

Traditional unit testing is case-based:

Test Case for max():

```
@Test
public void maxTest() {
  assertEqual(10, max(3, 10));
  assertEqual(10, max(10, 10));
}
```

- So, what does `max` really do?
  - Pick the right-hand side argument?
  - Always return 10?

Unit Testing

Traditional unit testing is case-based:

Test Case for max():

```
@Test
public void maxTest() {
  assertEqual(10, max(3, 10));
  assertEqual(10, max(10, 10));
}
```

- So, what does `max` really do?
  - Pick the right-hand side argument?
  - Always return 10?

Unit Testing

Traditional unit testing is case-based:

Test Case for max():

```
@Test
public void maxTest() {
  assertEqual(10, max(3, 10));
  assertEqual(10, max(10, 10));
}
```

- So, what does `max` really do?
  - Pick the right-hand side argument?
  - Always return 10?

Unit Testing

Traditional unit testing is case-based:

Test Case for max():

```java
@Test
public void maxTest() {
  assertEqual(10, max(3, 10));
  assertEqual(10, max(10, 10));
}
```

- So, what does `max` really do?
  - Pick the right-hand side argument?
  - Always return 10?

Unit Testing

We might add more cases:

```
@Test
public void testAdd() {
    Fraction a = new Fraction(1, 2);
    Fraction b = new Fraction(2, 3);

    assertFraction(1, 1, a.add(a));
    assertFraction(7, 6, a.add(b));
    assertFraction(7, 6, b.add(a));
    assertFraction(4, 3, b.add(b));

    Fraction f1 = new Fraction(Integer.MAX_VALUE - 1, 1);
    Fraction f2 = Fraction.ONE;
    Fraction f = f1.add(f2);
    assertEquals(Integer.MAX_VALUE, f.getNumerator());
    assertEquals(1, f.getDenominator());
    // ...
```

But...

How many cases do we need?

Can we learn anything useful about the behaviour from reading the tests?

Can I reuse my tests, or do I have to test `Integer.max` and `Double.max` separately?

But...

How many cases do we need?

- Rougly twice as many as you can think of... [Myers79]

Can we learn anything useful about the behaviour from reading the tests?

Can I reuse my tests, or do I have to test `Integer.max` and `Double.max` separately?

But...

How many cases do we need?

- Rougly twice as many as you can think of... [Myers79]

Can we learn anything useful about the behaviour from reading the tests?

- Probably not, but who reads tests anyway?

Can I reuse my tests, or do I have to test `Integer.max` and `Double.max` separately?

Axioms to the Rescue!

What are the fundamental properties of `max`?

$$\forall a, b : \max(a, b) == \max(b, a)$$

$$\forall a : \max(a, a) == a$$

$$\forall a, b : \max(a, b) >= a \wedge \max(a, b) >= b$$

$$\forall a, b : \max(a, b) == a \vee \max(a, b) == b$$

Axiom-Based or Property-Based Testing: Generate lots of values for a, b, and check that the axioms hold.

Axioms as Parameterised Tests

Axioms for Max

```
public void maxAxioms(TotalOrder<T> a, TotalOrder<T> b) {
  assertEquals(max(a, b), max(b, a));

  assertEquals(max(a, a), a);

  assertTrue(max(a, b) >= a && assertTrue(max(a, b) >= b));

  assertTrue(max(a, b) == a || assertTrue(max(a, b) == b));
}
```

(In pseudo-Java)

QuickCheck

- You provide a specification in the form of properties or axioms
- Automatically generates random data to exercise your axioms
- You can specify custom data generators
- You can check the distribution of your test data, classify your test cases and collect statistics about what's going on
- Highly popular with Haskell programmers!

Axioms as Parameterised Tests

Axioms for Max

```haskell
import Test.QuickCheck

prop_max1 a = max a a == a
  where types = a::Int

prop_max2 a b = max a b == max b a
  where types = a::Int

prop_max3 a b = max a b >= a && max a b >= b
  where types = a::Int

prop_max4 a b = max a b == a || max a b == b
  where types = a::Int
```

(In Haskell)

QuickCheck

Running QuickCheck

```
Main> quickCheck prop_max1
OK, passed 100 tests.

Main> quickCheck prop_max2
OK, passed 100 tests.

Main> quickCheck prop_max3
OK, passed 100 tests.

Main> quickCheck prop_max4
OK, passed 100 tests.
```

Bad Max

Axioms for MyMax

```
import Test.QuickCheck

mymax a b = b

prop_mymax1 a = mymax a a == a
  where types = a::Int

prop_mymax2 a b = mymax a b == mymax b a
  where types = a::Int

prop_mymax3 a b = mymax a b >= a && mymax a b >= b
  where types = a::Int

prop_mymax4 a b = mymax a b == a || mymax a b == b
  where types = a::Int
```

What Happens?

Running QuickCheck

```
Main> quickCheck prop_mymax1
OK, passed 100 tests.

Main> quickCheck prop_mymax2
Falsifiable, after 0 tests:
-2
-3

Main> quickCheck prop_mymax3
Falsifiable, after 0 tests:
1
-2

Main> quickCheck prop_mymax4
OK, passed 100 tests.
```

Testing with Concepts and Axioms

That's better...

But there's still some things to consider. How to make tests that are

- Reusable – build advanced specs from fundamental ones
- Generic – use the same axioms for int, real, number, ...

Introduction to Concepts

- Concepts are a way to specify interfaces and behaviour in Magnolia
- A `concept` consists of
  - types
  - operations
  - axioms
- A concept is essentially an algebraic specification
  - (Rewriting and optimisation)
  - Use in axiom-based testing
- Terminology is from Tecton (1981); similar feature was rejected from C++ 2011 (but we also have a library that provides C++ concepts)

A Concept is...

...a set of types, a set of operations and a set of axioms:

Concept Semigroup

```
concept Semigroup = {
  type T;
  function binop(a:T, b:T) : T;

  axiom associative (a:T, b:T, c:T) {
    assert binop( a, binop(b,c) ) == binop( binop(a,b), c);
  }
};
```

A concept is an interface only – no definitions are allowed.

Building Concepts

Concept Monoid

```
concept Monoid = {
  type T;
  function star(a:T, b:T) : T;
  use Neutral[binop => star, neutral => one];
  use Semigroup[binop => star];
};
```

Large concepts are built from small ones.

Building Concepts

Concept Numbers

```
concept Numbers = {
  /** The type of the numbers. */
  type Number;

  use UnitRing [ T => Number];
  use PartialOrder [ E => Number ];

  /** For numbers, minus one is less than zero and zero is less than one. */
  axiom zero_vs_one () {
    assert !(zero() <= uminus(one()));
    assert !(one() <= zero());
  }
};
```

`Numbers` is built on 15 other concepts (often reused several times);
`BoundedInteger` uses 35; arrays use 35-45 (depending on array kind)

Axioms

Axioms

```
axiom associative (a:T, b:T, c:T) {
  assert binop( a, binop(b,c) ) == binop( binop(a,b), c);
}

axiom hashing(a:Hashable, b:Hashable) {
  assert a == b => hash(a) == hash(b);
}
```

- universally quantified over parameters
- `assert` gives the actual axiom (multiple allowed)
- can use usual logic operators

Satisfaction

The satisfaction statement connects specification with implementation:

My integer implementation behaves as a bounded integer:

```
satisfaction boundedInteger32_is_BoundedInteger
   = boundedInteger32 models BoundedInteger;
```

- Renaming maps between implementation and specification names

```
satisfaction myAssocList_is_Dictionary
   = myAssocList models Dictionary[Dict => AssocList];
```

- Syntactic requirements are checked statically
- Semantic requirements / axioms are checked by testing and/or verification

Concepts as Specifications

- A concept can be seen as an algebraic specification
- We can have many implementations/programs that implement the specification
- Specification is done by relating the behaviour of operations
  - Not by listing particular inputs and outputs,
  - nor by listing pre- and postconditions
- A complete specification is not always necessary or desirable:
  - You can do useful testing with what you've got
  - You can refine a specification in a new concept
  - Error behaviour (or may not) may be better left undefined

Concept-Based Testing

The basic idea:

- Treat axioms as test oracles
  - Boolean functions that test the implementation given some data
- Feed generated test data to the oracles
  - You must supply a data generator
- For every implementation:
  - Call full test or individual tests
- All the paperwork should be handled automatically
  - tracking errors, axiom coverage, data distribution, ...

Generating Test Data

(We have this for C++, but not yet for Magnolia)

- Use random testing, specific data points or a combination
- Generators return sets of test data for a type
  - Construct using default constructor
  - List of predefined data
  - Term generator, run random functions to construct data
  - Multiple generators can be combined

Writing Axioms

A rule of thumb for writing axioms is

1. Divide functions into constructors and non-constructors
2. Write axioms for every constructor combined with every non-constructor

E.g., for a dictionary/hash map, with operations `contains`, `isEmpty`, `get`, `create` and `put`, we have constructors `create` and `put`. We then need to specify:

- `contains`, `isEmpty`, `get` applied to a new `Dict`
- `contains`, `isEmpty`, `get` after `put`

But you may want to leave the specification incomplete

- E.g., leaving `get(create(), k)` undefined

How's This Different From Pre/Post Conditions?

- You can easily specify relationships:

```
axiom hashing(a:Hashable, b:Hashable) {
  assert a == b => hash(a) == hash(b);
}
axiom pushPop(s:Stack, e:Element) {
  assert pop(push(e, s)) == e;
}
```

- Good for generic code
  - No need to specify details you don't know yet
    - Can connect `push` and `pop` without going via type invariant
  - Can specify requirements for parameters
- Preconditions still needed for partial functions
- Assertions/invariants still useful in algorithms / data structures

Integration of Implementations

We can also do interesting stuff with integration. For example,

- I have a hash table (basically, give me an array and a hash function, and I'll give you a dictionary)
- It only works if you provide a key type with a `hash` function

How to test?

- I can test the `hash` function in isolation
- I can find a suitable key type by searching for implementations that satisfy the `Hashing` concept, and test all of them [or vice versa]

Example Concept: Dictionary

```
concept Dictionary = {
  type Dict; type Key; type Val;

  function create() : Dict;
  function put(d:Dict, k:Key, k:Val) : Dict;
  function get(Dict, Key) : Val;
  predicate contains(d:Dict, k:Key) ;
  predicate isEmpty(d:Dict);

  axiom dict1(d:Dict, k:Key, v:Val) {
    assert get(put(d, k, v), k) == v;
    assert contains(put(d, k, v), k);
  }
  axiom dict2(d:Dict, k:Key, l:Key, v:Val, w:Val) {
    if(k != l)
      assert get(d, k) == get(put(d, l, w), k);
  }
}
```

Example Concept: Dictionary

```
concept Hash = {
  type Hashable;
  type HashVal;
  function hash(a:Hashable) : HashVal;

  axiom hashing(a:Hashable, b:Hashable) {
    assert a == b => hash(a) == hash(b);
  }
}
```

Conclusion – Benefits

- Build a library of reusable specifications
  - Less chance of making mistakes
- More general than unit testing
  - You'll test things you didn't think of
  - Can also be done with disciplined use of unit tests, if no tool is available
- Integrates well with an interface orconcept-oriented method
  - Domain engineering, discovering concepts
  - Writing and specifying concepts
  - Writing and testing implementations
- Implement in different ways, specify and test in one way

More Info

- http://bldl.ii.uib.no/testing.html
  - Catsfoot – library for C++
  - JAxT – library for Java
- Algebraic Specification
  - Liskov & Guttag books
- Uses of concepts / algebraic specification: Sophus, MTL4, STL
- Specification-based Testing
  - QuickCheck, ASTOOT, JAX, JAxT, DAISTS, Daistish, CASCAT, ...