QCon London 2016

An Introduction to Property Based Testing

ELIGIBLE

Aaron Bedra
Chief Security Officer, Eligible
@abedra

Why do we test?

- To better understand what we are building
- To help us think deeper about what we are building
- To ensure the correctness of what we are building
- To help us explore our design*
- To explain to others how our code should work

How do we test?

- With compilers (type systems, static analysis, etc)
- Manual testing
- X-Unit style tests
- Property/generative based tests
- Formal modeling

How do we test?

- With compilers (type systems, static analysis, etc)
- Manual testing
- X-Unit style tests
- **Property/generative based tests**
- Formal modeling

What is it?

An abstraction

Property based testing eliminates the guess work on value and order of operations testing

Magic numbers

Instead of specifying how you specify what

Testing over time

When we start our test suite, things are usually easy to understand

```java
public class Basic {
    public static Integer calculate(Integer x, Integer y) {
        return x + y;
    }
}
```

```
public class BasicTest {
    @Test
    public void TestCalculate() {
        assertEquals(Integer.valueOf(5), Basic.calculate(3, 2));
    }
}
```

What other tests might we write for this code?

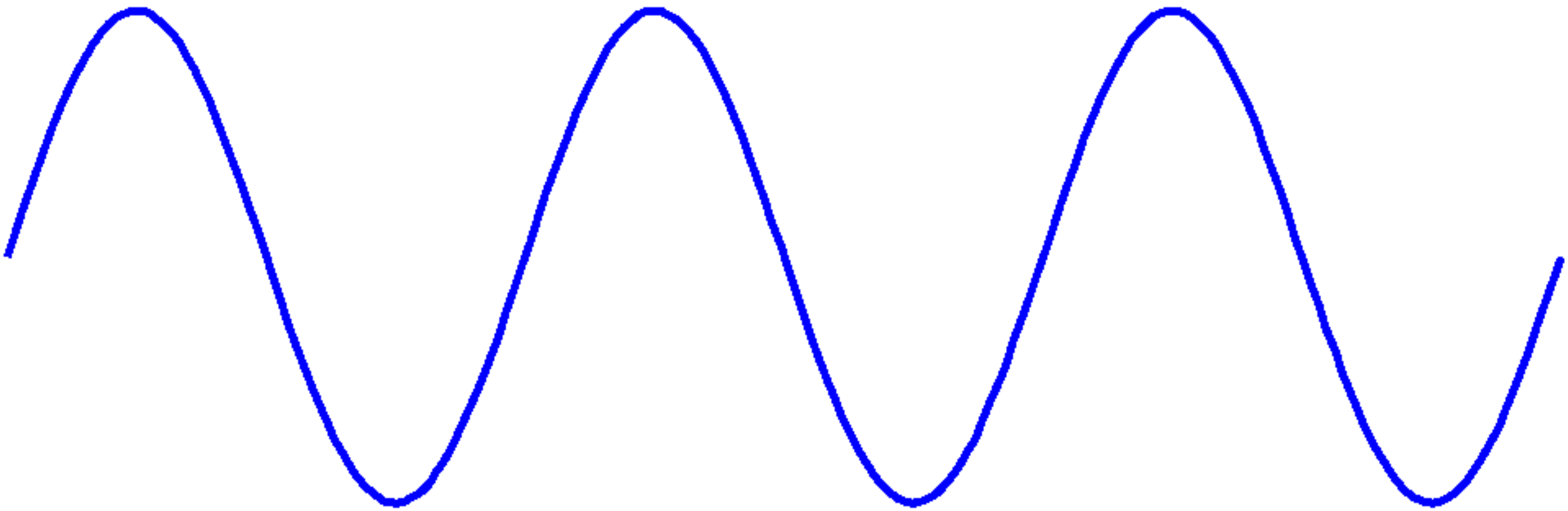
Like all programs we start simple

But over time things get more complicated

What happens when our simple calculate function grows to include an entire domain?

Our test suite will undoubtedly grow, but we have options to control the growth

And also maintain confidence in our tests



By changing our mental model just a bit we can cover much more ground

Let's revisit our basic example

```java
public class Basic {
    public static Integer calculate(Integer x, Integer y) {
        return x + y;
    }
}
```

But instead of a unit test,
let's write a property

```java
@RunWith(JUnitQuickcheck.class)
public class BasicProperties {
    @Property public void calculateBaseAssumption(Integer x, Integer y) {
        Integer expected = x + y;
        assertEquals(expected, Basic.calculate(x, y));
    }
}
```

```java
public class BasicTest {
    @Test
    public void TestCalculate() {
        assertEquals(Integer.valueOf(5), Basic.calculate(3, 2));
    }
}
```

```java
@RunWith(JUnitQuickcheck.class)
public class BasicProperties {
    @Property(trials = 1000000) public void
            calculateBaseAssumption(Integer x, Integer y) {
        Integer expected = x + y;
        assertEquals(expected, Basic.calculate(x, y));
    }
}
```

This property isn’t much different than the unit test we had before it

It’s just one level of abstraction higher

Let’s add a constraint to our calculator

Let’s say that the output cannot be negative

```java
public class Basic {
    public static Integer calculate(Integer x, Integer y) {
        Integer total = x + y;
        if (total < 0) {
            return 0;
        } else {
            return total;
        }
    }
}
```

java.lang.AssertionError: Property calculateBaseAssumption falsified for args shrunken to [0, -679447654]

Shrinking

```java
public class Basic {
    public static Integer calculate(Integer x, Integer y) {
        Integer total = x + y;
        if (total < 0) {
            return 0;
        } else {
            return total;
        }
    }
}
```

```java
@RunWith(JUnitQuickcheck.class)
public class BasicProperties {
    @Property public void calculateBaseAssumption(Integer x, Integer y) {
        Integer expected = x + y;
        assertEquals(expected, Basic.calculate(x, y));
    }
}
```

Now we can be more specific with our property

```java
@RunWith(JUnitQuickcheck.class)
public class BasicProperties {
    @Property public void calculateBaseAssumption(Integer x, Integer y) {
        assumeThat(x, greaterThan(0));
        assumeThat(y, greaterThan(0));
        assertThat(Basic.calculate(x, y), is(greaterThan(0)));
    }
}
```

```
java.lang.AssertionError: Property calculateBaseAssumption falsified for args shrunken to [647853159,
1499681379]
```

We could keep going from here but let’s dive into some of the concepts

Refactoring

This is one of my favorite use cases for invoking property based testing

Legacy code becomes the model

It’s incredibly powerful

It ensures you have exact feature parity

Even for unintended features!

Generators

You can use them for all kinds of things

Scenario

Every route in your web application

You could define generators based on your routes

And create valid and invalid inputs for every endpoint

You could run the generators on every test

Or save the output of the generation for faster execution

Saved execution of generators can even bring you to simulation testing

There are tons of property based testing libraries available

But this is a talk in a functional language track

So let’s have some fun

Let’s pretend we have some legacy code

Written in C

And we want to test it to make sure it actually works

But there are no quickcheck libraries available*

Warning! The crypto you are about to see should not be attempted at work

Caesar’s Cipher

Let's start with our implementation

```c
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

char *caesar(int shift, char *input)
{
  char *output = malloc(strlen(input));
  memset(output, '\0', strlen(input));

  for (int x = 0; x < strlen(input); x++) {
    if (isalpha(input[x])) {
      int c = toupper(input[x]);
      c = (((c - 65) + shift) % 26) + 65;
      output[x] = c;
    } else {
      output[x] = input[x];
    }
  }

  return output;
}
```

Next we create a new implementation to test against

```haskell
caesar :: Int -> String -> String
caesar k = map f
  where
    f c
        | inRange ('A', 'Z') c = chr $ ord 'A' +
                                  (ord c - ord 'A' + k) `mod` 26
        | otherwise = c
```

We now have two functions that “should” do the same thing

But they aren’t in the same language

Thankfully Haskell has good FFI support

```haskell
foreign import ccall "caesar.h caesar"
     c_caesar :: CInt -> CString -> CString

native_caesar :: Int -> String -> IO String
native_caesar shift input = withCString input $ \c_str ->
  peekCString(c_caesar (fromIntegral shift) c_str)
```

```
$ stack exec ghci caesar.hs caesar.so
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Main             ( caesar.hs, interpreted )
Ok, modules loaded: Main.
*Main> caesar 2 "ATTACKATDAWN"
"CVVCEMCVFCYP"
*Main> native_caesar 2 "ATTACKATDAWN"
"CVVCEMCVFCYP"
```

We can now execute our C code from inside of Haskell

We can use Haskell's quickcheck library to verify our C code

First we need to write a property

```haskell
unsafeEq :: IO String -> String -> Bool
unsafeEq x y = unsafePerformIO(x) == y

genSafeChar :: Gen Char
genSafeChar = elements ['A' .. 'Z']

genSafeString :: Gen String
genSafeString = listOf genSafeChar

newtype SafeString = SafeString { unwrapSafeString :: String } deriving Show
instance Arbitrary SafeString where arbitrary = SafeString <$> genSafeString

equivalenceProperty = forAll genSafeString $ \str ->
  unsafeEq (native_caesar 2 str) (caesar 2 str)
```

```haskell
unsafeEq :: IO String -> String -> Bool
unsafeEq x y = unsafePerformIO(x) == y

genSafeChar :: Gen Char
genSafeChar = elements ['A' .. 'Z']

genSafeString :: Gen String
genSafeString = listOf genSafeChar

newtype SafeString = SafeString { unwrapSafeString :: String } deriving Show
instance Arbitrary SafeString where arbitrary = SafeString <$> genSafeString

equivalenceProperty = forAll genSafeString $ \str ->
  unsafeEq (native_caesar 2 str) (caesar 2 str)
```

```haskell
unsafeEq :: IO String -> String -> Bool
unsafeEq x y = unsafePerformIO(x) == y

genSafeChar :: Gen Char
genSafeChar = elements ['A' .. 'Z']

genSafeString :: Gen String
genSafeString = listOf genSafeChar

newtype SafeString = SafeString { unwrapSafeString :: String } deriving Show
instance Arbitrary SafeString where arbitrary = SafeString <$> genSafeString

equivalenceProperty = forAll genSafeString $ \str ->
  unsafeEq (native_caesar 2 str) (caesar 2 str)
```

```haskell
unsafeEq :: IO String -> String -> Bool
unsafeEq x y = unsafePerformIO(x) == y

genSafeChar :: Gen Char
genSafeChar = elements ['A' .. 'Z']

genSafeString :: Gen String
genSafeString = listOf genSafeChar

newtype SafeString = SafeString { unwrapSafeString :: String } deriving Show
instance Arbitrary SafeString where arbitrary = SafeString <$> genSafeString

equivalenceProperty = forAll genSafeString $ \str ->
  unsafeEq (native_caesar 2 str) (caesar 2 str)
```

```haskell
unsafeEq :: IO String -> String -> Bool
unsafeEq x y = unsafePerformIO(x) == y

genSafeChar :: Gen Char
genSafeChar = elements ['A' .. 'Z']

genSafeString :: Gen String
genSafeString = listOf genSafeChar

newtype SafeString = SafeString { unwrapSafeString :: String } deriving Show
instance Arbitrary SafeString where arbitrary = SafeString <$> genSafeString

equivalenceProperty = forAll genSafeString $ \str ->
  unsafeEq (native_caesar 2 str) (caesar 2 str)
```

```haskell
unsafeEq :: IO String -> String -> Bool
unsafeEq x y = unsafePerformIO(x) == y

genSafeChar :: Gen Char
genSafeChar = elements ['A' .. 'Z']

genSafeString :: Gen String
genSafeString = listOf genSafeChar

newtype SafeString = SafeString { unwrapSafeString :: String } deriving Show
instance Arbitrary SafeString where arbitrary = SafeString <$> genSafeString

equivalenceProperty = forAll genSafeString $ \str ->
  unsafeEq (native_caesar 2 str) (caesar 2 str)
```

```
*Main> quickCheck equivalenceProperty
*** Failed! Falsifiable (after 20 tests):
"QYMSMCWTIXNDFDMLSL"
*Main> caesar 2 "QYMSMCWTIXNDFDMLSL"
"SAOUOEYVKZPFHFONUN"
*Main> native_caesar 2 "QYMSMCWTIXNDFDMLSL"
“SAOUOEYVKZPFHFONUN/Users/abedra/x“
```

```c
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

char *caesar(int shift, char *input)
{
  char *output = malloc(strlen(input));
  memset(output, '\0', strlen(input));

  for (int x = 0; x < strlen(input); x++) {
    if (isalpha(input[x])) {
      int c = toupper(input[x]);
      c = (((c - 65) + shift) % 26) + 65;
      output[x] = c;
    } else {
      output[x] = input[x];
    }
  }

  return output;
}
```

We’ve found a memory handling issue in our C code!

In reality there are more issues with this code, but our issue was quickly exposed

And easily reproduced

Wrapping up

Not all testing is created equal

You should use as many different testing techniques as you need

Remember to think about the limits of your tools

And use tools that help you achieve your results more effectively

And more efficiently

Questions?