Contents

Chapter 1

Foreword

When we wanted to run a two hour, self-paced, tutorial on Property-Based testing, we could not determine in advance what would work for the participants. Therefore we made too much. At the same time, we didn't create all the exercises we had in mind yet. We were not done learning, or teaching.

Before starting on this book, we learnt about property based testing by reading blog posts, attending a workshop, doing exercises on School of Haskell and applying it in our own projects.

As with regular Test Driven Development, we found the barrier to start applying property-based testing on a project can be high. Property-based testing is simple, but not easy. It takes time to learn. Your production code might be amenable to writing properties and generating random data. Writing these things takes time, so either you do it from the start of your project, or you train yourself to derive properties and generators for your own code quickly.

This book provides some of that training. We've created exercises that reflect what we need in our day-to-day work, with as little of the noise that comes with production software. At the same time, we try to minimise the gap between exercise and practice, so you can apply what you learnt quickly.

This book progresses from painting by numbers to more freeform exercises, with some asides on test suites and debugging. Do it at your own pace, in the order you feel like. We hope you experience the joy of properties!

Work in progress

This book is not finished. We've put this book on Leanpub so we could get feedback while we are writing it. It could be that no-one cares, that we have the

wrong exercises or questions, that we make mistakes (we sure do), or that there are topics we could cover that we have not thought of.

Changelog

15 December 2015

- Added Changelog section
- Upgraded instructions for sample project to use Stack
- Added more elaborate paint by numbers exercise to get you started
- Published an [initial solution for the gilded rose exercise](https://github.com/qwaneu/gilded-rose-haskell-solution.git)

Known defects

Some of the text describing the exercises is missing bits of code, or the code reflects the text of the javascript tutorial.

Chapter 2

What is Property Based Testing?

When we program test-first *example based testing*. We write an example of how a function or system under test behaves. Examples can communicate very well. It might be difficult to catch all corner cases. And with example based test, we try to find a minimal set of examples, to keep the maintenance burden low.

Example based testing provides just a few data points to capture the system under test. How do you know that you've actually caught those pesky corner cases in your examples?

Property based testing takes a different approach:

- you define properties - invariants of the system-under-test
- the property based testing library generate lots of random input data
- the system under test is run with this input data and the testing library verifies that the invariants hold under all inputs

Benefits

In our experience, property based testing can be beneficial in several cases:

- Characterization testing - understand existing libraries, quickly test a wide range of cases (e.g. numeric types or libraries to represent money).
- Testing validation logic
- Testing mappings / adapters

- Finding pesky corner cases
- When example based testing gets repetitive and/or you want to check many cases, and there is no obvious way to express the variations succinctly. Or when your examples don't express your intent very well, no matter how hard you try.

Furthermore, Property Based Testing drives your design, something it has in common with Test Driven Development (although it drives your design in a different way). Property based testing forces you to think about invariant properties of your code and capture these explicitly. Defining custom generators helps you get a better understanding of the preconditions of your code.

Property based testing is a relatively new practice. We are still learning about its applicability and usefulness.

Key concepts

Properties: invariants of the code under test; invariants should always be true under all valid inputs

Generators: A generator is a function that can generate arbitrary data of a specific type.

Shrinking: Shrinking is used when a counter example is found, to reduce the counter example to the smallest one that still fails the property

Arbitraries: An Arbitrary combines a generator with a shrinking function, and allows you to use random data in properties by only specifying the type you want, Haskell will then choose the right generator function automatically.

I want this in my favorite language!

Property based testing is available for most programming languages, like:

- Haskell: Quickcheck
- Scala: ScalaCheck. This one also has its own book: ScalaCheck, the definitive guide. We decided to write our own, because we want to go back to the source, and some of us work in Haskell.
- Java: Junit-QuickCheck
- Javascript: many libraries available, like *jsverify* and *claire*
- C# / .Net: FsCheck

- Elm: Elm-Check

We provide this tutorial in Haskell, because that is where it all started. There is also a javascript version of this tutorial with jsverify, which has not made it to book form yet (but could if you want).

Chapter 3

Getting Started

Before this chapter you need to...

Install the haskell tool Stack Know a little bit of Haskell

And maybe not even that. This chapter is painting-by-numbers. It explains how to run and set up an almost empty properties, it explains each line of code you need to write, and there are instructions on how to run the tests.

After this section you will...

- Be able to write a simple property using a built-in data type
- Wonder how to generate your own data.

Let's go

Open the sample project. The sample project contains one property that fails in a very obvious way. You can find it in:

```
src/Main.hs
```

Run it with

```
stack runghc src/Main.hs
```

If you want to divide an exercise in smaller modules, you can add them to hsprop.cabal later. For now, replace the code in src/Main.hs with the folowing:

```
-- We will explain why we need this extension later:
{-# LANGUAGE ScopedTypeVariables #-}
module GettingStarted where
```

We need the Test.QuickCheck module to generate random data and to calculate properties.

```
import Test.QuickCheck
```

Let's start with a simple example using some math functions, to introduce the basic concepts. An invariant of the square root and square functions is:

the square root of a number's square is equal to that number, in pseudocode:

```
sqrt( x * x ) = x
```

In an example based test approach, we would write a small number of examples showing that this holds. We now want to 'proof' that the mentioned invariant holds for all numbers.

Define the 'system under test'. We have a 'squared' function, and we don't care that much yet about what type its input 'x' is, so we leave it off for now.

```
squared x = x * x
```

We want to show that for all numbers, the following holds:

```
sqrt (squared n) == n
```

So we define a property stating just that. We add a type signature for a fitting number type, so that QuickCheck can choose what kind of values to generate. At this stage it would compile just fine without a type signature. Compilation will fail once you use it with quickCheck, because there are several Floating instances to choose from. We choose Double. y

```
prop_SquareRootOfNSquaredEqualsN :: Double -> Bool
prop_SquareRootOfNSquaredEqualsN n = (sqrt (squared n)) == n
```

Starting it with 'prop_' is more than just a naming convention, we will use this in later chapters to generate a test suite. You can load this into ghci and play around with it for some n. In this chapter we will show the quickCheck invocation from a

`mainN`

at each step, so you can more easily follow along in

`cabal repl`

.

And now comes the magic:

```
main0 = quickCheck prop_SquareRootOfNSquaredEqualsN
```

add it to

`runTests`

in

`GettingStarted.hs`

and use

`cabal -j build && ./dist/build/joy/joy`

.

or load it in cabal repl.

This will fail after some number of steps. For me it failed after three:

```
*** Failed! Falsifiable (after 5 tests and 1080 shrinks):
5.0e-324
```

What happened? QuickCheck generates input test data and checks if the property holds (returns true) for all the generated input values. If it finds a case for which the property does not hold, it fails, and returns the counter example.

How does it know to generate test data? QuickCheck will use the property's type to generate data. Default generators are available for many built-in types such as numbers, strings, and even functions. In this case it will generate Doubles, because that is what we specified in the type of the property.

QuickCheck tells us that the failing input is a very small number with mantissa 5 and exponent -324. What happens when we multiply and perform sqrt on such a small number?

Using QuickCheck for just a little bit, we realize that when we deal with floats, rounding is a problem. QuickCheck generates a really small number as value for ‘n‘, very close to zero and then runs out of precision when doing (sqrt (squared n)).

This is annoying. Let's work around it by making our own ‘sqrti‘ function that rounds to the nearest integer, and write a new property for which QuickCheck will generate only integers. Otherwise we could round inside the property, but that would be very messy.

```
sqrti :: Integral a => a -> a
sqrti = floor . sqrt . fromIntegral

prop_SqrtiOfSquaredNEqualsN :: Int -> Bool
prop_SqrtiOfSquaredNEqualsN n = (sqrti (squared n)) == n
```

Now it fails again:

```
*** Failed! Falsifiable (after 3 tests and 1 shrink):
-1
```

You notice that the output mentions tests and shrinks. The number of tests and shrinks you see will be different. When it finds a counterexample, QuickCheck shrinks the input so that you get the ‘smallest‘ input that fails. In this case the smallest number it can find. For lists it is the shortest list, etc.

The test fails because the invariant does not hold for negative numbers. We should restrict the generated input to natural numbers only (numbers ≥ 0). We can do this by changing the type of our generator to NonNegative a, in our case NonNegative Int. NonNegative is a Modifier:

https://hackage.haskell.org/package/QuickCheck-2.8.1/docs/Test-QuickCheck-Modifiers.html.

```
prop_SqrPositive :: NonNegative Int -> Bool
prop_SqrPositive (NonNegative n) = (sqrti $ squared n) == n

main3 = quickCheck prop_SqrPositive
```

This way we can communicate our intention for the property's input in its type.

We can also use generators explicitly inside the property to achieve a similar effect. You can find many available generators by browsing the documentation for the Gen module.

We could for instance say we are only interested in numbers between 1 and 100. For that we use the 'choose' function together with forAll. Since we made the type of sqrti a bit general, we have to specify what Integral we want, so we choose (n :: Int).

```
smallPositiveInteger = choose (1,100)

prop_SqrSmallInt = forAll smallPositiveInteger $ \n ->
  (sqrti (squared n)) == (n :: Int)

main4 = quickCheck prop_SqrSmallInt
```

There is one subtlety we did not show: if you use forAll, the type of your property changes. It no longer results in a Bool, but in a Property. We can declare it like this:

```
prop_SqrSmallInt :: Property


prop_SqrPositiveForGreaterEqualsZero n =
  n >= 0 ==> (sqrti (squared n)) == (n :: Int)
```

Here we show how to turn this into a small test suite by hand.

```
runTests = do
  quickCheck prop_SquareRootOfNSquaredEqualsN
  quickCheck prop_SqrSmallInt
  quickCheck prop_SqrPositive
  quickCheck prop_SqrtiOfSquaredNEqualsN
  quickCheck prop_SqrPositiveForGreaterEqualsZero
```

This acts as a nice summary for what we have just covered. We have written a deceptively simple function that, when it comes to testing, has a few edge cases we had to think about. Writing QuickCheck properties forced us to think about the type of inputs we can deal with and the constrainst to apply to that input. We have also played with a few ways to express constraints for a property.

Writing a test suite like this is boring. In the next seciont we will generate a test suite using Template Haskell.

Chapter 4

Debugging

Out of the box QuickCheck does not give that much feedback about what fails. It will show the input values for the property that failed, but what happens inside that property is invisible.

We've seen in the introduction that we can use the '===' operator to at least see the expected and actual output inside a property. But what if we don't want to compare with equality, or if we want to see some other values in our property or the code under test.

We will import Haskell's Debug.Trace module, so we can print values to the console, even though we are otherwise operating without IO.

```
module Debugging (runTests) where
import Test.QuickCheck
import Debug.Trace
```

Using standard haskell debugging

If you are experienced in debugging haskell programs, you can skip to the next section, otherwise it is a good idea to follow the example below, so that you don't get stuck doing the more elaborate exercise.

On a failure, QuickCheck only shows the resulting value, not the input. If you want to see all the inputs and results, use traceShow inside the property. This can be very useful when developing new properties, or understanding a regression.

We use the same squared function as in the introduction:

```
squared :: (Num x) => x -> x
squared x = x * x
```

This is the property we had. We'll use this below to illustrate some other forms of debugging as well.

```
prop_TraceSqrNEqualsN :: Double -> Property
prop_TraceSqrNEqualsN n = (sqrt $ squared n) === n
```

We add the debugging code in a where block, so we can easily reuse or remove it:

```
prop_TraceSqrNEqualsNDebug :: Double -> Property
prop_TraceSqrNEqualsNDebug n = debugShow $ result === n
  where
    result = sqrt $ squared n
    debugShow = traceShow $ "input: " ++ show n
                          ++ " result: " ++ show result
```

This will produce a lot of output. Even if the property fails after just a few tests, this will also print for all the shrinks. Here is a tail of the test run for the above property:

```
"input: 1.0e-323 result: 0.0"d 1077.1 shrinks)...
"input: 0.0 result: 0.0"ts and 1078 shrinks)...
"input: 5.0e-324 result: 0.0"d 1078.1 shrinks)...
"input: 0.0 result: 0.0"ts and 1079 shrinks)...
Falsifiable (after 2 tests and 1079 shrinks):
5.0e-324
0.0 /= 5.0e-324
*** Failed! Falsifiable (after 2 tests and 1074 shrinks):
5.0e-324
```

Playing with the number of test cases

By default it is configured to generate 100 test cases. You can specify a different number using quickCheckWith, here with a the property we just defined.

```
tenTests = quickCheckWith stdArgs { maxSuccess = 10 }
  prop_TraceSqrNEqualsNDebug
```

Expecting failure

We can mark tests as expectFailure, so a property suite as a whole will pass. This comes in handy for writing quickCheck tutorials - some of our properties should fail.

```
shouldFail = expectFailure prop_TraceSqrNEqualsN
```

The result is a success, and the output shows that the property fails as expected, so there is no mystery here:

```
*Debugging> res <- runTests
+++ OK, failed as expected. Falsifiable (after 3 tests and 1083 shrinks):
5.0e-324
0.0 /= 5.0e-324
```

TODO explain use of sample to see what data gets generated.

Running only the failing property

We collect only the one property that should fail, to demonstrate that we can run it just like any other property:

```
runTests = quickCheck shouldFail
```

Chapter 5

Basic generators and Properties - Menu Sorting

Dear leanpub reader, this is an exercise Marc Evers and I recently developed for a live coding demonstration at the JFall 2015 conference. This exercise is an attempt to plug a hole in the beginning of the workbook, start with an exercise that takes a reader through the main concepts step by step. This was one of the learning points from the session we ran at the Joy of Coding conference. The JFall demo was in Javascript with JSVerify, the haskell version was developed on the side

In a way the exercise worked, as some of the attendees at JFall got the idea quickly.

Why is this 'From the cutting room floor'?

What I like about the exercise is that it allows us to explain various uses of generators, so that more complex exercises like Money become more approachable. QuickCheck told me a thing or two of what can be in primitive datatypes, e.g. Strings with strange unicode escape sequences, and rounding of floating point numbers is worse than I thought. In this exercise generating readable strings makes doing the exercise more pleasant.

What I don't like about the exercise is that it is not particularly motivating - it is not something I can easily relate to my practice -, and that, especially in Haskell, it is quite hard to make the implementation sufficiently different from the properties.

I have an idea for a better exercise (something with simple graphics and constraints, where the implementation is slightly more involved than just specifying the constraints), but I need to work it out. I'm sharing this as 'from the cutting room floor', so you might correct me if we should keep it. It will probably be removed in a later version.

Menu Sorting - What is in a String?

We want to develop a simple restaurant menu, that can be grouped by course, and sorted by dish. The purpose of this exercise is not to wow you with a clever implementation, but to guide you, in baby steps, through writing generators for random data.

```
-- We will explain why we need this extension later:
{-# LANGUAGE ScopedTypeVariables #-}
module MenuSorting where
```

We need the Test.QuickCheck module to generate random data and to calculate properties.

```
import Test.QuickCheck
```

We'll intermingle properties and code in this exercise so you can follow along. Start with a stubbed functions:

```
menuSort _ = []
```

Start with the sorting of items, we don't concern ourselves with menu items yet, let's just do strings.

We write out our generator as a separate function, so that it is very clear what the property is going to test. We'll show a more compressed notation later, we chose this order so as to remove as much magic as possible.

```
stringlyItems :: Gen [String]
stringlyItems = arbitrary

prop_itemsAreSortedByDish = forAll stringlyItems (\items -> allSorted (menuSort items))

allSorted :: [String] -> Bool
allSorted xs  = fst (sortedUpTo xs)


sortedUpTo :: [String] -> (Bool, String)
sortedUpTo xs =
  foldr (\current (sofar, previous) -> (sofar && (current > previous), current))
        (True, "")
        xs
```

Surprise, surprise, all 100 tests pass!. We only checked if the empty list our sort returns is sorted, which it of course is.

All our original items have disappeared, and the empty list is sorted. Let's write a second property

```
prop_sortsAllItems = forAll stringlyItems (\items -> (length items) === (length (menuSort items))
```

That is better, we now have a failing property as well.

```
runTests = do
  quickCheck prop_itemsAreSortedByDish
  quickCheck prop_sortsAllItems
```

Sorting with words

```
module MenuSorting2 where

import Test.QuickCheck
```

We want to implement menuSort, therefore we add an import for Data.List to the start of the file.

```
import Data.List (sort)
```

Our menuSort function now looks like this:

```
menuSort = sort
```

The property fails, because our property does not deal well with escape codes.

TODO print failing output in book.

Maybe we should make our lives easier for now, and restrict the generated strings to upper and lowercase letters.

We first build a letter generator, that gives us a lowercase or uppercase character, and then build a word generator on top of that.

We create a list of lower and uppercase letters by using a haskell range, and then use elements to generate a character from that list. We can then build up a word using listOf, and, renaming our stringlyItems, we can use listOf again to build a list of words.

Because the prelude already has a word function, we prefix our generator functions with 'gen'.

```
genLetter :: Gen Char
genLetter = elements (['a'..'z'] ++ ['A'..'Z'])

genWord :: Gen String
genWord = listOf genLetter

-- call this 'sentence' and add spaces in between to make it more interesting
-- and we get 'forAll sentence' which doesn't clash with prelude
-- and adds a step of interestingness above genWord.
-- possibly with frequency: 50% words, 50% spaces
-- we're not looking for the ultimate precision here, we don't care whether a sentence
-- other kinds of interpunction. double spaces or no spaces may appear as well, just a
genWords :: Gen [String]
genWords = listOf genWord

prop_itemsAreSortedByDish = forAll genWords (\items -> allSorted (menuSort items))

allSorted :: Ord a => [a] -> Bool
allSorted [] = True
allSorted [x] = True
allSorted (x:y:xs) = (x <= y) && allSorted (y:xs)
```

Because allSorted is quite involved, we add a trivial property to keep us honest. The first time we implemented it wrong, so it passed.

```
prop_sortsAllItems = forAll genWords (\items -> (length items) === (length (menuSort it
```

We've worked with strings so far, but we are really interested in Records. The allSorted property/testable looks like it might be useful with things that can be compared for their ordering What if we replace our String with an a that implements the Ord typeclass?

allSorted :: Ord a => [a] -> Bool

That compiles, and the tests still pass. Now we are almost good to go for the next step. Can we refactor more?

It is a bit tedious to keep rewriting prop_itemsAreSortedByDish all the time. It could easily test various sorting functions if we can pass them in.

how about:

```
prop_itemsAreSortedBy sorter genCollection =
  forAll genCollection (\items -> allSorted (sorter items))
```

```
-- |since we are only sorting strings so far, maybe we should be honest
-- in the name of our property
prop_stringsAreSorted = prop_itemsAreSortedBy menuSort genWords
```

prop_itemsAreSortedBy is our first reusable property. We pass in the function we want to test, and the generator we want to use.

```
runTests = do
  quickCheck prop_itemsAreSortedByDish
  quickCheck prop_sortsAllItems
  quickCheck prop_stringsAreSorted
```

Sorting and sampling food records

In this section we'll structure our menu, and experience sampling food with our generators.

```
module MenuSorting3 where

import Test.QuickCheck

import Data.List (sortBy, sort, groupBy)
```

Let's do another iteration and expand our generators and properties to support the domain model we actually want. We define a record for food items. The fields in the items are still Stringly typed, but we'll adapt our generators to generate better fitting strings than in the previous round. We derive 'Eq' so we can easily compare Food in properties, and 'Show' so we can see the actual items when our properties fail.

```
data Food = Food { course :: String, dish :: String }
              deriving (Show, Eq)
```

Let's make a generator for Food based on the letter and word generators we had before. We generate a word for a course an a word for a dish, and then construct an arbitrary Food based on that.

```
genLetter :: Gen Char
genLetter = elements (['a'..'z'] ++ ['A'..'Z'])
```

```
genWord :: Gen String
genWord = listOf genLetter

genFood :: Gen Food
genFood = do
  aCourse <- genWord
  aDish <- genWord
  pure (Food aCourse aDish)

genMenu :: Gen [Food]
genMenu = listOf genFood
```

TODO sample genLetter and genFood

We wanted to sort food by course. This is also a good time to give our menuSort function a better name. We'll use sortBy and a function to compare two food items.

```
sortFoodByCourse :: [Food] -> [Food]
sortFoodByCourse = sortBy compareFood

compareFood :: Food -> Food -> Ordering
compareFood rhs lhs = compare (course lhs) (course rhs)
```

For compareFood we compare the courses, by reusing the compare function defined for strings.

– Move this down to the refactoring part, for this piece rewrite the allSorted function using (course x) <= (course y), so it looks differently on the surface.

How do we fit our food into the allSorted function? We define oure own compare function. We just gave allSorted an Ord constraint. We implement compare in the Ord typeclass for Food by reusing our compareFood function.

```
instance Ord Food where
  compare = compareFood
```

allSorted and prop_itemsAreSortedBy are the same as in the previous round.

```
allSorted :: Ord a => [a] -> Bool
allSorted [] = True
allSorted [x] = True
allSorted (x:y:xs) = (x <= y) && allSorted (y:xs)

prop_itemsAreSortedBy sorter genCollection = forAll genCollection (\items -> allSorted
```

For genCollection we can use genFoods, and we have a sorter defined.

```
prop_sortFoodByCourse = prop_itemsAreSortedBy sortFoodByCourse genMenu
```

This property passes.

We rewrite our sanity check to use the new sorting function

```
prop_sortsAllItems = forAll genMenu (\items -> (length items) ===
                                      (length (sortFoodByCourse items)))
```

This also passes. There are, however, some things that should give us pause.

We don't know what's in our Food. Let's sample.

We use the compare function for our functionality as well as for our property. So what are we testing?

The other one is our testdata. Although we now have a Food record, when we use QuickChecks sample function in ghci to generate some sample data we see this:

```
ghci> sample genFood
Food {course = "", dish = ""}
Food {course = "k", dish = "yZ"}
Food {course = "ZEBM", dish = ""}
Food {course = "S", dish = ""}
Food {course = "jN", dish = ""}
Food {course = "vsfNiGZlRt", dish = "QZerby"}
Food {course = "f", dish = "e"}
Food {course = "vpYS", dish = "gRxdRs"}
Food {course = "B", dish = "fIfeOxroPyjasoLy"}
Food {course = "JGRwEsDtD", dish = "bXY"}
Food {course = "onuYhj", dish = "MxXtJUqIMfQpKwib"}
```

– conditional properties / more precise generation is useful We generate empty strings for courses as well as dishes, and the names for both are not exactly pretty.

– I'd drop the groupBy part - the property becomes a lot more complicated, Finally, we have the Food sorted per course, but when we make a menu, we want to group them by course. Perhaps that will help us define a more meaningful property.

We have duplication between properties and implementation. We'll refactor that only after we're happy with our generators.

What happens when you can't generate enough data?

We could filter out the empty courses in our properties. Let's see how this looks in the sortBy property.

We have to break open our general property for sorted items and modify that to filter out non-empty courses.

```
nonEmptyCourse :: Food -> Bool
nonEmptyCourse food = not (null (course food))

prop_nonEmptyitemsAreSortedBy sorter genCollection = forAll genCollection
   (\items -> allSorted (sorter (filter nonEmptyCourse items)))

prop_sortFoodByNonEmptyCourse = prop_nonEmptyitemsAreSortedBy sortFoodByCourse genMenu
```

The property passes, and we are not testing for empty courses anymore. Or are we? QuickCheck is not aware that we are removing some items to check, so it reports 100 tests passed, of which an unknown number are empty strings.

The code for the properties is not very nice either. We are getting more nesting, and the general SortedBy property we had can now only be applied to Food.

We can use a conditional property, so that Quickcheck can tell us how many items have been discarded. We use the ==> arrow to add a condition before a property.

We move the forAll around and split our property in two functions to get better feedback from the compiler while writing the property.

```
-- ? type Menu = [Food]
prop_conditionalSorted :: ([Food] -> [Food]) -> [Food] -> Property
prop_conditionalSorted sorter items =
  all nonEmptyCourse items ==> allSorted (sorter items)
prop_conditionalSortedFood =
  forAll genMenu (prop_conditionalSorted sortFoodByCourse)
```

When you run this, you'll still see "Ok, passed 100 tests.". It might look like nothing has changed. Remove the 'not' in nonEmptyCourse and see what happens now.

This is what I got:

```
*** Gave up! Passed only 44 tests.
```

Generating only empty courses by filtering out the non-empty ones is too expensive, and QuickCheck gives up. For the condition we had, it seems good

enough and just keeps on generating until it has found a hundred menus without empty courses.

We tend not to use conditional properties much, forAll and custom generators. We find forAll and custom generators easier to write and read.

What if we generate only non-empty words? Lets say we generate words of three or more letters, and at the same time limit their length - a menu will look representative without infinitely long strings. We can use vectorOf to generate vectors of a certain length, and determine the length with choose.

```
genMenuWord :: Gen String
genMenuWord = do
  length <- choose (3,30)
  vectorOf length genLetter
```

When we run 'sample' again, we get strings like "ffX" as the shortest, that is looking a bit more representative already.

Since courses are more limited than dishes, we could simplify that further, and make it more representative at the same time. We use the 'elements' generator we saw earlier.

```
genCourse :: Gen String
genCourse = elements ["appetizers", "starters", "main course", "desert"]
```

Refactoring property and implementation

Now that Food implements the Ord typeclass, we don't need sortBy to sort the food, once again we are back to having an alias to a library function.

```
sortFood :: Ord a => [a] -> [a]
sortFood = sort


-- |since we are only sorting strings so far, maybe we should be honest
-- in the name of our property
--prop_stringsAreSorted = prop_itemsAreSortedBy menuSort genWords

runTests = do
  quickCheck prop_sortFoodByCourse
  quickCheck prop_sortFoodByNonEmptyCourse
  quickCheck prop_sortsAllItems
  quickCheck prop_conditionalSortedFood
```

Chapter 6

Modelling Money

In this exercise, we are going to build a class representing Money, step by step. Representing money in software might sound easy, but it is actually non trivial and is often done wrong. Doing it wrong can result in small or big rounding errors or just losing a few millions on a bad day...

Getting started

Open the Money.hs file, and add the quickcheck boilerplate, and import fmap and sequential application from Control.Applicative:

```
-- for Arbitrary Amount and Currency, and to add Amounts
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
module Money where
import Test.QuickCheck
-- we use expectFailure to ensure incomplete examples fail
import Test.QuickCheck.Property ((===), expectFailure)
import Test.Hspec (describe, it)
import Language.Haskell.TH
-- to generate Arbitrary instances
import Control.Applicative ((<$>), (<*>))
```

Create a GADT to represent money. It is constructed with an amount and a currency. This is a good moment to think of the types of amount and currency. (TODO from quickCheck manual: use newtypes ??)

We learnt in the introduction that QuickCheck could easily force us to change our mind about the types that we use. Primitive obsession (using Strings, Ints and other primitive data types instead of types we define ourselves) is painful

enough. Lets keep our options open and define an Amount as an Int, meaning a number in cents. For the purpose of this exercise we don't want to get into rounding errors again, we've been there already today. The Currency will be represented as a String.

```
newtype Amount = Amount {amountValue :: Int}
                   deriving (Arbitrary, Num, Eq, Show)
newtype Currency = Currency { currencyValue :: String}
                     deriving (Eq,Show)
```

For Amount we use GeneralizedNewTypeDeriving, so we can derive an Arbitrary instance to generate arbitrary amounts. The Eq and Show instance are needed so our properties can compare amounts, and quickcheck can show us the failures.

Currency has a string on the inside. We could of course do this with a proper datatype, but for this tutorial the strings make it a bit more interesting, because we don't want arbitrary strings, we want strings representing a currency.

```
data Money = Money { currency :: Currency
                   ,amount :: Amount
                  } deriving (Eq, Show)
```

Choosing explicit data types also will help us generate very specific arbitrary data with QuickCheck.

Currency Properties

When we add two amounts of the same currency, we get a new amount. Our property needs two money objects, so we define a property function with two arguments:

```
prop_moneyOfSameCurrencyAddsUp :: Money -> Money -> Property
prop_moneyOfSameCurrencyAddsUp m1@(Money _ (Amount a1))
                               m2@(Money _ (Amount a2)) = (a1 + a2) === total mtotal
  where total (Right money) = amountValue $ amount money
        -- TODO Magic number
        total (Left msg)    = -4242424242424242
        mtotal = addMoney m1 m2
```

Not very pretty, and we are ignoring the currency!

We could run this property by writing:

```
runTest = quickCheck prop_moneyOfSameCurrencyAddsUp
```

This is sweet and simple, but breaks down when you write two or three properties. If one of them fails, it is necessary to scroll, and guess which property failed. I tend to use HSpec, an RSpec inspired test framework. It forces me to name properties descriptively, clearly shows which properties failed, and how many out of the total. It also allows us to run only a selection of tests, so we can focus what to work on.

```
currencySpec = describe "Currency" $
    it "Adds up money of the same currency" $
      property $ \m1 m2 -> prop_moneyOfSameCurrencyAddsUp m1 m2
```

I end to extract properties into functions so the level of indentation in hspec stays manageable, and the types can be clearly annotated. Your mileage may vary.

When you run this, it looks like:

```
Currency
  Adds up money of the same currency FAILED [1]

Failures:

  literate-haskell-lib/Money.lhs:77:
  1) Currency Adds up money of the same currency
       Falsifiable (after 1 test):
       Money {currency = Currency {currencyValue = "GBP"}, amount = Amount {amountValue = 0}}
       Money {currency = Currency {currencyValue = "USD"}, amount = Amount {amountValue = 0}}
       0 /= -4242424242424242

Randomized with seed 1227787702

Finished in 0.0005 seconds
1 example, 1 failure
```

Exercise: After you've defined the generator for currencies and Arbitrary Money below, get rid of the magic number, and find a way to express the constraint that two Moneys should be of the same currency.

Now we have to generate arbitrary money. We'll do this in small steps, a bit smaller than we would do in production code, so you can see another example of a custom generators as well as Arbitrary instances for more complex data types.

QuickCheck can generate random amounts for us without our intervention. But we don't want random strings for currencies! We can generate the currency using the ‘elements‘ generator:

```
currencies :: Gen Currency
currencies = elements(map Currency ["EUR", "USD", "GBP"]);
```

How do we generate Money instances? We define an instance of the Arbitrary typeclass and create an implementation for its' arbitrary generator. Arbitrary is what provides shrinking behaviour etc, but the only function we really have to implement is arbitrary.

We use the Applicative instance for Arbitrary to construct Money, because there is no dependency between currency and arbitrary. We could have used the Monad instance here (as you will see in the original quickCheck paper), but that would have been overkill. The Appliccative instance makes it almost look like we are just constructing a record the regular way. This makes it quite easy to create Arbitrary instances for all our domain objects.

```
instance Arbitrary Money where
  arbitrary = Money <$> currencies <*> arbitrary
```

We could also define an Arbitrary for currency like this,

```
instance Arbitrary Currency where
  arbitrary = elements(map Currency ["EUR", "USD", "GBP"])
```

or simply reusing the 'currencies' generator we defined above. When do you think deriving an Arbitrary is more useful than a generator function? When do you think defining a generator function is more useful than an Arbitrary instance? Does it make sense to create new data types specifically so that we can make Arbitraries?

Lots of choice here. Play around with them and see if and how they influence the design of your properties and production code.

Now we can generate Money objects!

Implement behaviour for Money

Now add the behaviour that Money can only be added to money of the same currency. Express that different currencies are not supported (e.g. by using an Either). Make sure your property handles this correctly.

```
newtype ErrorMessage = ErrorMessage { errorMsg :: String } deriving (Eq,Show)

addMoney :: Money -> Money -> Either ErrorMessage Money
-- your implementation here
```

Add a new property that checks that adding is not possible whenever the currencies are different.

Exercise: Money allocation

We want to be able to allocate money: divide it in equal parts, without money getting lost in rounding. An example: dividing EUR 100 by 3 should yield:

```
[EUR 33, EUR 33, EUR 34]
```

We are assuming whole numbers for now.

Define a property that states that no money gets lost when dividing an amount.

Note that we probably should make the integer assumption of the amounts in the example explicit in the types.

Exercise: Converting from string

We want to be able to create money objects from strings; we support EUR and USD. Some legal values:

- 100 (default is EUR)
- EUR 10
- USD 300
- USD 40- (negative value)
- EUR 10.000 (. as thousand separator for euros)

And some illegal values:

- USD 4.999 (USD has , as thousand separator and . as cents separator)
- 1,000 (illegal because EUR is default and EUR uses . as thousand separator)
- HFL 50 (illegal currency)
- EUR 10bla

In our domain model, we work with valid objects. This part focuses on processing and validation outside, 'untrusted' data that is transformed into domain objects (e.g. data coming in from a web form). We want this transformation to be robust, so that it can handle any data and only results in valid domain objects when it is able to (and we don't need to do defensive programming in our domain).

There are actually two concerns we need to address:

- Given the input string is valid, will our code create a valid Money object with the correct contents? (correctness)
- It should not throw unexpected exceptions, be only defined for a part of the input, or create invalid Money objects (robustnesss). It should either return an error message, or a money object.

Let's capture these in properties.

Define a parse function that returns a Either ErrorMessage Money.

```
parseMoney :: String -> Either ErrorMessage Money
--TODO your implementation here

parseMoney s = Right $  Money (Currency "EUR") (Amount $ read s)

propNumberParsesAsEUR :: Int -> Property
propNumberParsesAsEUR n = parseMoney (show n) === Right (Money (Currency "EUR") (Amoun

parseCurrencySpecs = describe "Parsing money" $
    it "Defaults to EUR" $ property propNumberParsesAsEUR
```

We can start with either correctness or robustness first. It would be interesting to try both approaches and see whether and how it drives you to different design decisions.

Correctness

Given correct input, the conversion should produce valid Money objects. How would you define this as property?

Write an arbitrary that creates valid input strings. You can again use smap to create valid strings from primitive values.

Hints and tips: - If you let your arbitrary generate a tuple (array) of an input string together with the corresponding amount and currency values, it will be easier. for your property function to check correctness:

- Take baby steps; start e.g. with regular input with a currency; add support for the default currency; add support for thousand separators. - Make sure your test fails for the right reason at every step.

Robustness

Given any input, the conversion should always produce either a valid Money object or a validation error:

```
{ money: ... } or { error: 'some message' }
```

Define a property that captures this.

Write an arbitrary that creates all kinds of input strings. Is the 'asciistring' arbitrary suited for this? Why (not)?

Corner cases that almost look like valid input are particularly interesting, like "EUR 12jsde", "HFL 30", and "300 EUR". Adapt your arbitrary so that it will generate cases like these.

Refine your conversion function and make sure the correctness property is also satisfied.

Hints and tips: - QuickCheck provides the 'oneOf' generator that combines two generators into one that generates values taking values from both generators.

Reflection

How did defining the properties and the arbitraries influence you in thinking about the problem?

Where would you like to put the property logic? To what extent is it test code, to what extent should/could it be part of production code?

What happens if you would start with robustness first and then correctness? Do the exercise again, but start with robustness and see if this leads you to different design decisions.

We end with the usual boilerplate to generate tests with TemplateHaskell.

```
currencySuite = do
  currencySpecShouldFail
  parseCurrencySpecs

currencySpecShouldFail = describe "Currency" $
    it "Adds up money of the same currency" $
      property $ expectFailure $ \m1 m2 -> prop_moneyOfSameCurrencyAddsUp m1 m2
```

Chapter 7

A suite of properties

If you've done example based testing, you are probably familiar with the concept of a test suite. While QuickCheck can be used with hspec and tasty, in the following chapters we will use a bit of template haskell to collect our properties for us, so we don't need to introduce a test framework as well.

The module Test.QuickCheck.All contains a few '*checkAll' functions to automatigically run all the properties in a module. This way we can quickly create a suite of properties.

We'll demonstrate it with quickCheckAll, you can look up the others in the module documentation. To let quickCheckAll generate code, we need the TemplateHaskell language extension.

```
{-# LANGUAGE TemplateHaskell #-}
module PropertySuite where
```

We've seen Test.QuickCheck, and Test.QuickCheck.Property before. We need the Template Haskell (TH) module in addition to the language pragma.

```
import Test.QuickCheck
import Language.Haskell.TH
import Test.QuickCheck.All (quickCheckAll)
import Test.QuickCheck.Property ((===))
```

We'll introduce two small properties to show we can run them. Their names start with prop_ so quickCheckAll can find them.

```
prop_reverse xs = (reverse (reverse xs)) === xs
```

```
prop_associative :: Int -> Int -> Int -> Property
prop_associative x y z = ((x + y) + z) === (x + (y + z))
```

The following code has to go at the bottom of your module, otherwise Template-Haskell can not find all the functions.

```
return [] -- needed in GHC 7.8 series, not after
runTests = $quickCheckAll
```

There are a few variants of *CheckAll that you may find useful, See the Test.QuickCheck.All documentation. So now you know what the Template-Haskell and the bit of magic at the bottom of the following chapters are for.

Chapter 8

A vending machine with a coin box

In this exercise, we are focusing on a vending machine. In particular the cash register part of such a machine.

Start with modelling a coin box, in which customers can insert different coins and after making a selection, the appropriate amount is checked out. In the end, the coin box should also correcly handle change: any money left over is returned to the customer, using appropriate coins.

Getting started

Let's start simple. We want to create a Coinbox with 'insert' and 'checkout' functions. We can model the coin box as having two collections of coins: the 'inbox' (coins inserted but not yet processed) and the 'vault' (coins collected by the coin box after doing checkouts. The coin box keeps the coins until it is emptied by the owner. The coins from the vault can also be used to return the correct change, depending on what types of coins are needed.

We can create representations of all kinds of different coins, but it is sufficient for now to just model two or three (like 10 cents and 50 cents).

Start with a few simple invariants, like:

- the total value of the coins in the inbox should always be >= 0
- the total value of the coins in the vault should always be >= 0

Create a new module with the necessary imports. Create the properties one by one, first letting them fail. Write the simplest version of the Coinbox that satisfies a property.

Mutating state: doing checkouts

Now proceed with implementing checkouts. For now, assume that you do a checkout of all the coins present in the inbox.

First, a property concerning checkouts: first, a checkout empties the inbox. Write a property for this, see it fail for the right reason.

Second, a checkout should not modify the total value of the coins in the inbox and vault.

Write the property, see it fail for the right reason. Implement the checkout function for Coinbox.

Observe closely what happens. Do you get failures? Are they what you expect? If not, what happens?

Some tips

- It is useful to derive Show for Coinbox and Coins.
- The === operator will show the left- and right-handside on failure.

See the Debugging section for more tips.

Inserting coins

Let's add an insert function for the customer to insert one coin.

What property/properties can you define regarding inserting coins?

Code these properties and implement the insert. Make sure all other properties keep on being satisfied!

Possible next steps

If you want to continue with this exercise, your next steps could be:

- Conditional checkout: pass a price to the checkout function and only check out if the inbox contains a sufficient amount of money.

- Refine the checkout so that a specific amount is checked out and the remaining amount is returned as change. How will you make the coin box return the right coins?

Chapter 9

Legacy code - The Gilded Rose Kata

Legacy Code is great, because it can provide us with a free test oracle.

A test oracle can be defined as:

> a function which, given a program, P, can determine, for each input, x, if the output from P is the same as the output from a 'correct' version of P.

(William E. Howden 1987, p 43). See [A note on test oracles]{#test-oracle}

Legacy code can function as the 'correct' version in the above definition. The test oracle being the existing code. What we'll demonstrate here, is how to use existing, messy code and build a new version of that code that does everything the old code did, only in a more understandable way.

As Emily Bache writes in Writing Good Tests for the Gilded Rose Kata:

> When you design a test suite you have two main aims – to help you understand what the code should do, (and what it does now), and protection from regression failures when you update it. It can be a bit tricky to do both with the same test suite. If you focus solely on describing the requirements in an executable way, you tend to miss edge cases and there are gaps in the regression protection. If you focus only on regression protection, you'll spend time analysing the edge cases, and measuring code coverage to see how well you're doing, but the test cases can become quite hard to read and understand.

With QuickCheck we can have our cake, and eat it too. Use one property to describe all the edge cases, by comparing our shiny new code's output to the legacy code's output. Therefore our test cases don't become hard to read and understand.

We can add more properties as we wish, to document our understanding, but the first one will have us covered already.

In this chapter, you'll work through an exercise, we'll show you how to measure coverage, and we'll show a possible solution and some of the
trade-offs we made.

Exercise

Check out the starting project . This contains the legacy code, an empty file for QuickCheck properties and an empty file for a new implementation. The README.md contained in the project has instructions on how to build and run it.

We're not going in to how to break dependencies to get to a point where one can test something in isolation. The legacy code we have here, from the GildedRose kata has one redeeming quality: it is already purely functional.

Let's get started. We need to import QuickCheck, our existing code and it is probably best to make a new module for our new code. The extra constraint we give ourselves is to not touch the existing code. Normally, I'd be tempted to dive in and start refactoring, at least the bits in GildedRose that are type safe.

TODO

- Decide wether to elaborate on test oracle, or just keep it to the original definition.

What did you learn?

I learnt that rewriting a small, but very messy program, can be done safely with quickcheck. Assuming that the orginal program is correct, or at least good enough. Given the original code is quite messy, it is safe to assume it is not fulfilling all of its intentions. At least the new version can give us some more insight in what the missing pieces could be.

Chapter 10

Chapters we'll write if you want us to

This is a book in progress. We have a few exercises you can do now, and some we might write in the future.

The order here is not necessary the order they will apear in the book - the worked out examples will probably form the second half of the book.

Worked through examples for each exercise

Some of the later workbook chapters will be quite open - A few questions, some links, some hints. We believe it would be good to have fully worked out examples, explained step by step. This could help you when you get stuck, or see another perspective. We will not claim our answer is 'the best' by any stretch of the imagination. If you don't find this valuable let us know - if some of you do, and some don't, the answers might be better off in a separate booklet.

Lazyness to the max - Auto-generalize your code and properties with quickspec

QuickSpec is a package based on QuickSpec that will run a set of properties and report back to you which 'laws' your code might obey. Have you ever wondered whether a piece of code you encounter might be a Monoid or some other standard typeclass, but didn't have the energy to check? I have. We'll work through a little example and use Conal Elliots' Checkers library so we don't have to write the properties ourselves.

The benefit we'd hope to get from this is that users of your code can often program to more general interfaces, thus lowering the barrier to writing reusable libraries.

Testing concurrency

Ok, you can't really test concurrency, or at least prove that your application is behaving correctly, but it is possible to find defects. Inspiration for an hands-on exercise will be drawn from this talk by John Hughes, one of QuickCheck's inventors.

Specify a DSL by example

Thinking of a A free monad to build an interpreter with some interesting properties.

Legacy code and coverage

Property-based testing works great when you have a test-oracle and some data you can generate. We've tried QuickCheck on Gilded Rose Kata and it was quite interesting. Legacy code can be an area where looking at code coverage can safely be considered useful Haskell has excellent profiling and coverage facilities built-in.

We use the existing code as a test oracle, and develop some interesting properties as we work on writing new, clean, code to match the old hard to understand code's behaviour.

Testing with State - REST APIs

QuickCheck is great when you've got pure code. There is also support for testing stateful code. We'll look at writing some tests for a REST API, e.g. to check REST endpoints against cross-site-scripting attacks.

An added benefit of this, is that you can sneak Haskell and QuickCheck in on existing projects - write tests for a REST API in another language. We've done this with great success with tests in Ruby in the past.

Using QuickCheck for end-to-end tests with Selenium Webdriver

We found a blog post explaining how to do this, and for that use case it sounded useful. We have written example-based tests with hsWebDriver and could not yet see how it would be useful for us. In other words: We are lacking in imagination here. Provide us with another application and we might just write this chapter. In general writing end-to-end tests with Haskell is pretty sweet, if the tests are well-typed, than it is safe to refactor towards a readable Domain Specific Language that describes your system from an end-users' perspective.

Chapter 11

Useful reading for QuickCheck

Original papers

- QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs, Koen Claessen and John Hughes. In Proc. of International Conference on Functional Programming (ICFP), ACM SIGPLAN, 2000. Very readable, many examples and references, and still useful after 15 years.

- Testing IO actions with QuickCheck by Koen Claessen and John Hughes

- Introduction to QuickCheck on the HaskellWiki has a useful list of references as well.

Original tutorial and API Docs

- QuickCheck: an Automated Testing Tool for haskell This is the reason we are using NewType extensively in our tutorial.

- The QuickCheck package is on Hackage. The documentation for Test.QuickCheck contains some examples, and since Test.QuickCheck re-exports many of the underlying modules, this is a good place to start exploring the API in depth.

Integratiion with other test frameworks

Hspec

The Hspec github page has a working example of a quickcheck property inside an hspec describe block.

Tasty

Same for the Tasty github page .

Up and downsides of both

Upsides:

- One test suite to rule them all
- Readable explanation / reports
- Fail fast behaviour - quickCheckAll runs all tests even when one fails. E.g HSpec can stop after a testsuite has failed.

Downsides:

- May need a separate testsuite anyway, because a QuickCheck suite runs a lot more tests than a regular one and may therefore be slow.
- Duplication. Name of property and the string in the testsuite describing it may be earily similar. Hspec and Tasty both need a couple of lines per property. It can be compressed by e.g. using mapM_, but then the reporting advantage also disappears.

Integrating in a test suite probably works best if properties are one-liners.

It might be worth bundling a few quickCheck tests in a `runTests` function, e.g. by using `quickCheckAll` and integrating that into a test suite as opposed to individual properties.

Chapter 12

Things to explore

Reusing properties across typeclasses

- Verifying Typeclass Laws in Haskell with QuickCheck by Austin Rochford. Shows how to write reusable properties and generators for typeclasses.
- The Checkers package by Conal Elliot , provides arbitrary instances and generator combinators for common data types.

Using Property based tests for stateful applications

Redesign stateful code so we can test it as pure functions

- Purifying code using Free Monads by Gabriel Gonzalez

Shows how to transform a program containing some input and output into a Free Monad, and then use an interpreter in pure code together with QuickCheck to validate some assumptions.

Testing stateful code as is

- Testing a web application with hspec (http://looprecur.com/blog/testing-a-web-application-with-hspec/) by Adam Baker has a small example of using QuickCheck with monadic IO.
- Testing IO Actions with monadic quickcheck on StackOverflow has a few more.

- QuickCheck and WebDriver by Christian Brink. Check that a property holds over a number of web pages. In this case a form of decoration.

Testing legacy code

- QuickCheck as a test set generator on the Haskell Wiki. You can use just the Arbitraries and generators to generate data for just about anything. Lecacy code, or, we use it to test migrations - generate a data structure for one version, save it, migrate, and check the results with a regular unit test.

Chapter 13

Essays including QuickCheck as part of a wider discussion

- Software Testing From the Perspective of a Hardware Engineer by Dan Luu . About combining random tests with code coverage, and some things you'll miss even then.

Chapter 14

Appendix - From the cutting room floor

A note about test oracles

For the definition of a test oracle I quoted (William E. Howden 1987, 43). A book to which I don't have access. I found this quote in (Peters and Parnas 1994, 16).

The term test oracle was apparently coined by William E. Howden .

The oldest reference I could find was (William E Howden 1978, p1), where various kinds of test oracles are described in the introduction. I emphasized the definition I believe is most suitable for what we just did:

> In order to use testing to validate a program it is necessary to assume the existence of a test oracle which can be used to check the correctness of test output. The most common kind of test oracle is *one which can be used to check the correct- ness of output values for a given set of input values.* Other kinds of test oracles can be used to check the correctness of value traces of selected program variables. Formally defined oracles may consist of tables of values, algorithms for hand computation, or formulas in the predicate calculus. Informally defined oracles are often simply the assumed ability of the programmer to recognize correct output.

A recent discussion on oracles (Bolton 2015) seems to suggest that oracles originally were about correctness only

> This is why, in Rapid Software Testing, we say that an oracle is *a means by which we recognize a problem when it happens during testing.*

Emphasis theirs. To me, the "the assumed ability of the programmer to recognize correct output is also about finding things out while working.

It looks like the authors are in violent agreement in at least one place: Bolton :

> Although we can demonstrate incorrectness in a program, we cannot prove a program to be correct.

Howard:

> If a computer program has a small finite domain, and a test oracle is available which can be used to check the correctness of output values, then the use of testing to prove correctness is straightforward. When the domain is large it is necessary to rely on mathematical theory.

They cite from (Miller and Howden 1981) :

> The use of testing requires the existence of an external mechanism which can be used to check test output for correctness. This mechanism is referred to as the test oracle. Test oracles can take on different forms. They can consist of tables, hand calculated values, simulated results, or informal design and requirements descriptions.

So maybe over time the emphasis on correctnes grew, I don't know. To me it comes across as a straw man[^A straw man is a common form of argument and is an informal fallacy based on giving the impression of refuting an opponent's argument, while actually refuting an argument which was not advanced by that opponent.], at least for some of their points.

Michael Boltons' piece does have some interesting links to take into account when thinking of whether an oracle is suitable or not.

Wrote that oracles are about problems, not correctness and list some things to keep in mind while using oracles, as well as some counter-indications.

Chapter 15

References

Graphics credits

Checkmarks on cover: „Checkmark“ von Nobbler 76 - Eigenes Werk. Lizenziert unter Gemeinfrei über Wikimedia Commons - https://commons.wikimedia.org/wiki/File:Checkmark.svg#/media/File:C

Bolton, Michael. 2015. “Oracles Are About Problems, Not Correctness « Developsense Blog.” http://www.developsense.com/blog/2015/03/oracles-are-about-problems-not-correctness/.

Howden, William E. 1978. “Theoretical and Empirical Studies of Program Testing.” *Software Engineering, IEEE Transactions on*, no. 4: 293–98.

Howden, William E. 1987. *Functional Program Testing and Analysis*. McGraw-Hill Series in Computer software Engineering and Technology. New York: McGraw-Hill.

Miller, Edward, and William E Howden. 1981. *Tutorial, Software Testing & Validation Techniques*. IEEE Computer Society Press.

Peters, Dennis, and David L Parnas. 1994. “Generating a Test Oracle from Program Documentation: Work in Progress.” In *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 58–65. ACM.