# Exploring type-directed, test-driven development using FizzBuzz

Franklin Chen
http://franklinchen.com/

Pittsburgh Scala Meetup
May 15, 2014

# Types versus tests?

- I don't like flame wars.
- I like to use every available helpful tool when programming.

# Types are surprisingly underrated

- Archaic, painful type systems stuck in the 1970s: C, C++, Java.
- Reasonable type systems: Haskell, OCaml, F#, Scala.

# Purpose of this talk

- A case study in using a type-directed mindset, in conjunction with a test-driven process.
- Why `FizzBuzz`? Easy to understand and modify.
- Only a tiny taste of how to use types.
- I will use only one “m”-word.

# The basic user story

Given:

- User story for the FizzBuzz problem.

  *Write a program that prints the numbers from 1 to 100. But for multiples of three, print "Fizz" instead of the number. And for the multiples of five, print "Buzz". For numbers which are multiples of both three and five, print "FizzBuzz".*

- It could change.
- There could be more user stories.

# Implicit assumptions?

Not everything is fully specified:

- "Print" to where? To terminal window? To Web page?
- How should the output be formatted?
- Does the order of output matter?

# Make initial decisions

Judgment calls:

- Don't over-engineer up front.
- Do make note of assumptions, in case they change.

We will initially assume:

- Command-line application with text output.
- Each number's output on its own line.
- Output lines in order, not interleaved.

# Use a walking skeleton

Do not waste time doing just the simplest possible thing.

- End-to-end with a walking skeleton.
- We know we will *generalize* to output to an arbitrary output stream.
- We know we will *generalize* to input of any range of integers.

Starter Scala code:

```
object Main {
  def main(args: Array[String]): Unit = {
    runToStream(1, 100, Console.out)
  }

  def runToStream(start: Int, end: Int,
    stream: PrintStream): Unit = {
    ???
  }
}
```

# Unimplemented code

`???` is actually Scala code:

- Added to Scala for version 2.10 in January 2013.
- When executed, throws `NotImplementedError` exception.
- Convenient for TDD: do not need to write pointless "stub" code that might result in accidents.

# First test: an acceptance test

Top-down, outside-in design.
Use Scala testing framework specs2, which supports:

- Unit specifications.
- Acceptance specifications.

```
class MainSpec extends Specification { def is = s2"""
  ${Main.runToString(1, 16) ==== ‘expected for 1 to 16‘}
  """
}
```

# Expected output, as string

```
  val `expected for 1 to 16` = """1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
"""
```

# Continuous compilation and testing

I use SBT, the build tool supporting Scala, Java, etc:

- Continuous, incremental compilation! Source file changes trigger smart recompilation.
- Continuous, incremental testing! Source file changes trigger rerun of tests depending on changed code.

# The test does not compile

```
$ sbt
> ~testQuick

...src/test/scala/com/franklinchen/MainSpec.scala:10:
val runToString is not a member of
object com.franklinchen.Main
  ${Main.runToString(1, 16) ==== ‘expected for 1 to 16‘}
          ^
one error found
(test:compile) Compilation failed
```

# Make the test compile and fail

Create the skeleton for `Main.runToString`:

```
def runToString(start: Int, end: Int): String = {
  ???
}
```

The test now compiles, but fails:

```
MainSpec
  x Main.runToString(1, 16) ==== ‘expected for 1 to 16
  an implementation is missing (Main.scala:31)
```

# Start filling in code

```
def runToString(start: Int, end: Int): String = {
  val outputStream = new ByteArrayOutputStream
  val stream = new PrintStream(outputStream)
  runToStream(start, end, stream)
  outputStream.toString
}
```

- `runToString` just uses `runToStream`.
- It compiles.
- Top-down: still have not implemented `runToStream`.
- Did not bother to write a test for `runToString`.
- Candidate for refactoring into utility function; but now is not the time.

# Outside-in: toward a FizzBuzz unit

Use of *types* to drive *functional programming*:

```
def runToStream(start: Int, end: Int,
  stream: PrintStream): Unit = {
  (start to end).
    map(FizzBuzz.output).
    foreach(stream.println)
}
```

Follow the pipeline of types[1]:

- `(start to end): Seq[Int]`, where `Seq[_]` is a type constructor that given a type `A`, returns a type of `Seq[A]`.
- We need to implement `FizzBuzz.output: Int => String`.
- Mapping through `Int => String` gives `Seq[String]`.
- More generally, for any types `A` and `B`, mapping `Seq[A]` through `A => B` gives `Seq[B]`.
- `foreach` on a `Seq[A]` through `A => Unit` gives `Unit`.

[1] simplified for clarity

# Starting the FizzBuzz module

```
object FizzBuzz {
  type Outputter = Int => String

  val output: Outputter = ???
}
```

- Acceptance test drove the need for a unit.
- Implementing outside-in drove the discovery of a needed type, `Int => String`.
- For clarity, name the type (`Outputter`).

Time to write unit tests!
Start with an *example-based* test:

```
class FizzBuzzSpec extends Specification { def is = s2"""
  ${FizzBuzz.output(15) ==== "FizzBuzz"}
  """
}
```

# Property-based vs. example-based tests

Use ScalaCheck to write *property-based* tests:

- Automatically randomly-generated tests (default to 100).
- Completely *type-driven*: here, random `Int` values are generated.

```
class FizzBuzzSpec extends Specification
    with ScalaCheck { def is = s2"""
  ${‘Multiple of both 3 and 5 => "FizzBuzz"‘}
  ${‘Multiple of only 3 => "Fizz"‘}
  ${‘Multiple of only 5 => "Buzz"‘}
  ${‘Not a multiple of either 3 or 5 => number‘}
  """

  def ‘Multiple of both 3 and 5 => "FizzBuzz"‘ =
    prop { i: Int => (i % 3 == 0 && i % 5 == 0) ==>
      { FizzBuzz.output(i) ==== "FizzBuzz" }
    }
```

# Property tests, continued

(continued)

```
  def `Multiple of only 3 => "Fizz"` =
    prop { i: Int => (i % 3 == 0 && i % 5 != 0) ==>
      { FizzBuzz.output(i) ==== "Fizz" }
    }

  def `Multiple of only 5 => "Buzz"` =
    prop { i: Int => (i % 3 != 0 && i % 5 == 0) ==>
      { FizzBuzz.output(i) ==== "Buzz" }
    }

  def `Not a multiple of either 3 or 5 => number` =
    prop { i: Int => (i % 3 != 0 && i % 5 != 0) ==>
      { FizzBuzz.output(i) ==== i.toString }
    }
}
```

# Booleans are evil!

Do not enter the maze of twisty little conditionals, all different!

- Conditions can be arbitrary: depend on *any* combination of data.
- A computation leading to a Boolean value by essence loses information about the original data.
- Multiple conditions: combinatorial explosion (two conditions led to four cases).
- Possibly overlapping conditions: order dependency subtleties.
- Possibly duplicated checking of the some condition.
- Copy-and-paste eyesore of three almost-identical property-based tests and a fourth slightly different one.
- **No help from type system to catch wrongly written sets of nested, combined conditionals**.

# Thought-driven development

The core of software development: not *coding*, but *thinking*!
Observations:

- Treating `"FizzBuzz"` as an atomic unit ignores information: it is really a concatenation (*sum*) of `"Fizz"` and `"Buzz"`.
- The output string, when at least *some* condition holds, is the sum of output strings for each condition, if we consider a failed condition to map to an *empty* string.
- Special case: when *no* condition holds, by the rule above, we would expect an empty string; but instead, we should get the numerical string.

What now?

- Stop thinking, just write some code now that we have tests?
- Think more?

A judgment call.

# Naive implementation of `FizzBuzz`

Write the implementation straight from the tests, using pattern matching:

```
val output: Outputter = { i =>
  (i % 3 == 0, i % 5 == 0) match {
    case (true, true) => "FizzBuzz"
    case (true, false) => "Fizz"
    case (false, true) => "Buzz"
    case (false, false) => i.toString
  }
}
```

This works.

# Acceptance test passed!

```
MainSpec
   + Main.runToString(1, 16) ==== ‘expected for 1 to 16
```

Are we done?

# Changing the user story

In the real world, we are *never* "done".
Our users are demanding the ability to customize `FizzBuzz`.

- Someone wants to choose two numbers other than 3 and 5.
- Someone is offended by the words "fizz" and "buzz" and wants to choose different words to be output.

# Refactoring with types

- Add new tests.
- Change types and code just enough to make the new tests *type check*.
- Refactor the original `FizzBuzz` to use the new APIs.
- The modified `FizzBuzz` must still pass the old tests.

# More features means adding types

Changes to Main:

```
  def runToStream(start: Int, end: Int,
    stream: PrintStream): Unit = {
    (start to end).
      map(Defaults.fizzBuzzer).
      foreach(stream.println)
  }
```

Changes to FizzBuzz:

```
type WordMap = OrderedMap[Int, String]
type Compiler = WordMap => Outputter

val compile: Compiler = { wordMap =>
  i => ???
}
```

# Extract defaults

```
object Defaults {
  val wordMap: FizzBuzz.WordMap =
    SortedMap(3 -> "Fizz", 5 -> "Buzz")

  val fizzBuzzer: FizzBuzz.Outputter =
    FizzBuzz.compile(wordMap)
  }

  val oldFizzBuzzer: FizzBuzz.Outputter = { i =>
    (i % 3 == 0, i % 5 == 0) match {
      case (true, true) => "FizzBuzz"
      case (true, false) => "Fizz"
      case (false, true) => "Buzz"
      case (false, false) => i.toString
    }
  }
}
```

# Generalizing the `FizzBuzz` problem

A deep fact of mathematics and computer science: solving a *harder* problem is often the path toward understanding and efficiently solving an easier problem!
Consider a `FizzBuzzPop` problem:

- Test multiples of 3, 5, 7.
- Output a total string combining segments of `"Fizz"`, `"Buzz"`, `"Pop"`, or a numerical string if not a multiple of any of 3, 5, 7.

Example:

- `21` should result in `"FizzPop"`.

# Document all our thoughts

To clarify our thinking, write some new tests, with no intention of working on making them pass yet:

```
def is = s2"""
...
${Defaults.fizzBuzzPopper(2) ==== "2"}
${Defaults.fizzBuzzPopper(21) ==== "FizzPop"}
${Defaults.fizzBuzzPopper(9) ==== "Fizz"}
${Defaults.fizzBuzzPopper(7) ==== "Pop"}
${Defaults.fizzBuzzPopper(35) ==== "BuzzPop"}
"""
```

Since `FizzBuzzPop` is popular, put it into `Defaults`:

```
val fizzBuzzPopper: FizzBuzz.Outputter =
  FizzBuzz.compile(
    SortedMap(3 -> "Fizz", 5 -> "Buzz", 7 -> "Pop")
  )
```

# More than 2 is “many”!

- We do not want to hard-code some new solution for the case of 3 words.
- Instead, solve the general case of “many” (2 or more) words, with user-specified mapping of numbers to words.

# Even more types

Try to represent every idea as a type:

- Each configuration number (such as 3, 5, or 7 in `FizzBuzzPop`) should result in a *rule* that can be applied to any input number to get a string.
- Once we have a bunch of rules, we can apply them all to the input, then combine the partial results.

```
type Rule = Int => String
type Rules = Seq[Rule]
type RuleBuilder = ((Int, String)) => Rule

val buildRule: RuleBuilder = { case (n, word) =>
  i => ???
}
```

# Type-directed programming: solving a jigsaw puzzle

```
val buildRule: RuleBuilder = { case (n, word) =>
  i => if (i % n == 0) word else ""
}

val compile: Compiler = { wordMap =>
  val numbers: Iterable[Int] = wordMap.keys
  val pairs: Seq[(Int, String)] = wordMap.toSeq
  val rules: Rules = pairs.map(buildRule)

  i =>
  val words: Seq[String] = rules.map { rule => rule(i) }
  val combined: String = words.mkString
  if (combined.isEmpty) i.toString else combined
}
```

# Generalize our tests

A sample generalized test over user word choices:

```
def ‘Arbitrary word fizzBuzzers on a multiple of 3‘ =
  prop { (fizz: String, buzz: String) =>
    val fizzBuzzer = FizzBuzz.compile(
      SortedMap(3 -> fizz, 5 -> buzz)
    )
    prop { i: Int => (i % 3 == 0 && i % 5 != 0) ==>
      { fizzBuzzer(i) ==== fizz }
    }
  }
```

# Test failure reflecting poor use of types

Did you see this coming?

```
x Arbitrary word fizzBuzzers on a multiple of 3
A counter-example is ['', ''] (after 0 try)
A counter-example is '1029'
  (after 0 try - shrinked ('1138893201' -> '1029'))
'1029' is not equal to '' (FizzBuzzSpec.scala:58)
```

A good example of the usefulness of property-based testing!

- Empty "fizz" and "buzz" words are a strange corner case.
- Unexpected ambiguity: intended behavior was for a number to be output only if the given number is not a multiple of any of the factors, but 1029 is a multiple of 3.

# An empty string is not equivalent to no string

Presence of something "empty" is not equivalent to the absence of something (contrary to how some programming languages work).

- Problem: special case condition, testing for an empty string, conflated an empty combined string with "failed to be a multiple at all".
- Solution: add use of another type.

# Option[A]

Option[A] is one of two possibilities:

- None
- Some(a) for some value a of type A.

For example, Some("") is not the same as None:

```
val fizzFor1029 = Some("")      // multiple of 3
val buzzFor1029 = None          // not multiple of 5
val fizzbuzzFor1029 = Some("") // fizzed ""

val fizzFor2 = None             // not multiple of 3
val buzzFor2 = None             // not multiple of 5
val fizzbuzzFor2 = None         // not multiple of any
```

# Cleaning up the types

Change type `Rule`:

```
// old: type Rule = Int => String
type Rule = Int => Option[String]
```

Immediately get type errors:

```
found   : String
required: Option[String]
   i => if (i % n == 0) word else ""
                        ^

found   : String("")
required: Option[String]
   i => if (i % n == 0) word else ""
                                ^

found   : Seq[Option[String]]
required: Seq[String]
   val words: Seq[String] = rules.map { rule => rule(i) }
                                      ^
```

# Fix the type errors

```
val buildRule: RuleBuilder = { case (n, word) =>
  i => if (i % n == 0) Some(word) else None
}

val compile: Compiler = { wordMap =>
  val numbers: Iterable[Int] = wordMap.keys
  val pairs: Seq[(Int, String)] = wordMap.toSeq
  val rules: Rules = pairs.map(buildRule)

  i =>
  val words: Seq[Option[String]] =
    rules.map { rule => rule(i) }
  val combined: Option[String] = words.reduce(optionAdd)
  combined getOrElse i.toString
}
```

# Monoids

We implemented an "addition" for `Option[String]`:

```
val optionAdd:
  (Option[String], Option[String]) => Option[String] = {
  (a1, a2) => (a1, a2) match {
    case (Some(s1), Some(s2)) => Some(s1 ++ s2)
    case (Some(s1), None)     => a1
    case (None,     Some(s2)) => a2
    case (None,     None)     => None
  }
}
```

Monoid:

- There is an identity element.
- There is a binary associative operator.

More generally: `Option[A]` is a monoid if `A` is a monoid.

# Parallelism

- All code here using *map* can be parallelized[2].
- The one place where *reduce* is used can be parallelized because of the monoid property.

[2] Scala provides parallel collections

# Conclusion

- Tests are useful.
- Types are useful.
- Thinking is useful.

- Tests and types work great together.
- Try to encode your thoughts in types and tests.

All materials for this talk are available at https://github.com/franklinchen/talk-on-type-directed-tdd-using-fizzbuzz. The hyperlinks on the slide PDFs are clickable.