Exploring type-directed, test-driven development

A case study using FizzBuzz

Franklin Chen

http://franklinchen.com/

June 7, 2014

Pittsburgh TechFest 2014

Outline

- Introduction
- Original `FizzBuzz`
- `FizzBuzz` 2
- `FizzBuzz` 3
- Parallel `FizzBuzz`
- Conclusion

Goals of this presentation

- Give a taste of a type-directed, test-driven software development process.
  - See testing frameworks in action.
  - See types in action.
- Using `FizzBuzz` because:
  - The basic problem is easy to understand.
  - Modifications will be easy to understand.
  - You will see something surprising and cool!
- Encourage you to explore further.

What is test-driven development (TDD)?

- First, write a test case.
- Then, write code.
- Rerun the test.
- Repeat.

What is a type system?

- For this presentation: a syntactic method for proving the absence of certain program behaviors.
- Versus: runtime tag system.

"Debating" types "versus" tests?

- Let's not argue.
- Let's use both!

Crappy versus decent type systems

- Archaic 1960s-1970s-era type systems give types a bad reputation!
  - C, C++, Objective C
  - Java
- "Modern" 1980s-1990s-era type systems.
  - ML (Standard ML, OCaml, F#): I first used in 1994 (20 years ago)
  - Haskell: I first used in 1995
  - Scala: first released in 2004
  - Rust: not yet version 1.0
  - Swift: announced by Apple on June 2, 2014!

Original FizzBuzz problem statement

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".

Starter Scala code: main driver

Let's use Scala, a modern object-oriented and functional language.

```
object Main extends App {
  // Will not compile yet!
  runToSeq(1, 100).foreach(println)
}
```

- Type-directed design: separate out effects (such as printing to terminal) from the real work.
- Type-directed feedback: compilation fails when something is not implemented yet.

The joys of continuous compilation and testing

Let's use SBT, a build tool supporting Scala, Java, etc.

- Source file changes trigger smart recompilation!
- Source file changes trigger rerun of the tests that depend on changed code!

```
$ sbt
> ~testQuick
[info] Compiling 1 Scala source to ...
[error] ...Main.scala:16: not found: value runToSeq
[error]   runToSeq(1, 100) foreach println
[error]   ^
[error] one error found
[error] (compile:compile) Compilation failed
```

Fix compilation error using stub

```scala
object Main extends App {
  runToSeq(1, 100) foreach println

  def runToSeq(start: Int, end: Int): Seq[String] = {
    ???
  }
}
```

- Write desired type signature.
- `???` is super-convenient for stubbing.
  - From Scala standard library.
  - Just does `throw new NotImplementedError`.

Acceptance test (simplified)

Let's use specs2, a popular testing framework for Scala, Java, etc.

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

  val ‘strings for 1 to 16‘ = Seq(
    "1", "2", "Fizz", "4", "Buzz", "Fizz",
    "7", "8", "Fizz", "Buzz", "11", "Fizz",
    "13", "14", "FizzBuzz", "16"
  )
}
```

- A realistic acceptance test would involve I/O.
- Omitted in presentation to save time.

The test compiles, but fails

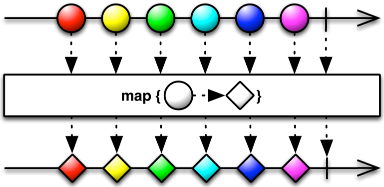
- Save brand new source file `MainSpec.scala`.
- Incremental compilation/testing continues.

```
Waiting for source changes... (press enter to interrupt)
[info] MainSpec
[info]   x Main.runToSeq(1, 16) ==== ‘strings for 1 to 16
[error]  an implementation is missing (Main.scala:19)
```

Outside-in: toward a FizzBuzz unit

Use types to assemble code like shapes in a jigsaw puzzle:

```
def runToSeq(start: Int, end: Int): Seq[String] = {
  (start to end) map FizzBuzz.evaluate
}
```



- `(start to end): Seq[Int]`, where `Seq[_]` is a type constructor that, given a type `A`, returns a type of `Seq[A]`.
- For any value of type `Seq[A]`, `map: (A => B) => Seq[B]`.
- So, we need to implement `FizzBuzz.evaluate: Int => String`.

Starting the FizzBuzz module

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

  val evaluate: Evaluator = { i =>
    ???
  }
}
```

- Acceptance test drives need for units.
- Outside-in implementation drives discovery of a type: `Int => String`.
- I like to write type synonyms as documentation:
  - `type FizzBuzzer = Int => String`.
  - I dislike comments. Comments are not executable and checkable, unlike types and tests.

First unit tests: example-based

```
class FizzBuzzSpec extends Specification { def is = s2"""
  ${FizzBuzz.evaluate(15) ==== "FizzBuzz"}
  ${FizzBuzz.evaluate(20) ==== "Buzz"}
  ${FizzBuzz.evaluate(6) ==== "Fizz"}
  ${FizzBuzz.evaluate(17) ==== "17"}
  """
}
```

The joy of property-based tests

Let's use ScalaCheck to write property-based tests.

```
class FizzBuzzSpec extends Specification
    with ScalaCheck { def is = s2"""
  ${`Multiple of both 3 and 5 => "FizzBuzz"`}
  """
  def `Multiple of both 3 and 5 => "FizzBuzz"` =
    prop { i: Int => (i % 3 == 0 && i % 5 == 0) ==>
      { FizzBuzz.evaluate(i) ==== "FizzBuzz" }
    }
}
```

- Type-driven: here, random `Int` values are generated.
- Automatically randomly generates tests for each property (defaults to 100).

Property-based tests, continued

The other three cases of interest:

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

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

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

A wrong and ugly solution

```scala
// Buggy and ugly!
val evaluate: Evaluator = { i =>
  if (i % 3 == 0)
    "Fizz"
  else if (i % 5 == 0)
    "Buzz"
  else if (i % 3 == 0 && i % 5 == 0)
    "FizzBuzz"
  else
    i.toString
}
```

```
[info] FizzBuzzSpec
[info]   x FizzBuzz.evaluate(15) ==== "FizzBuzz"
[error]  'Fizz' is not equal to 'FizzBuzz'
             (FizzBuzzSpec.scala:14)
```

Booleans are evil!

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.
- **No help from type system to catch wrongly written sets of nested, combined conditionals**.

Pattern matching organizes information

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

- Visual beauty and clarity.
- No ordering dependency.
- No overlapping.
- No duplicated conditionals.
- Type checker verifies full coverage of cases.

Example of non-exhaustive pattern matching

```scala
val evaluate: Evaluator = { i =>
  (i % 3 == 0, i % 5 == 0) match {
    case (true,  false) => "Fizz"
    case (false, true)  => "Buzz"
    case (true,  true)  => "FizzBuzz"
  }
}
```

```
[warn] ...FizzBuzz.scala:46: match may not be exhaustive.
[warn] It would fail on the following input: (false, false
[warn]     (i % 3 == 0, i % 5 == 0) match {
[warn]     ^
```

Acceptance test passed, finally

```
[info] MainSpec
[info]   + Main.runToSeq(1, 16) ==== ‘strings for 1 to 16
```

Are we done?

Adding new features

Client was pleased with our `FizzBuzz` solution.
In the real world, we are never "done". Client wants to:

- Specify two arbitrary divisors in place of `3` and `5` (such as `4` and `7`).
- Specify other arbitrary words in place of `"Fizz"` and `"Buzz"` (such as `"Moo"` and `"Quack"`).

Type-driven refactoring

Types make refactoring much more fun.

- Add new tests.
- Change types and code enough to make the new tests type check.
- Refactor the original code to use the new APIs.
- Keep passing the old tests.

More features means more types

Change Main.runToSeq driver:

```
def runToSeq(start: Int, end: Int): Seq[String] = {
  (start to end) map Defaults.fizzBuzzer
}
```

Add new types to FizzBuzz module:

```
type Evaluator = Int => String
case class Config(pair1: (Int, String),
                  pair2: (Int, String))
type Compiler = Config => Evaluator

val compile: Compiler = {
  case Config((divisor1, word1), (divisor2, word2)) =>
    {  i =>
      ???
    }
}
```

Extract original default configuration

```
object Defaults {
  val fizzBuzzerConfig: Config =
    Config(3 -> "Fizz", 5 -> "Buzz")

  val fizzBuzzer: Evaluator =
    FizzBuzz.compile(fizzBuzzerConfig)

  // Useful to keep old implementation
  val oldFizzBuzzer: Evaluator = { i =>
    (i % 3 == 0, i % 5 == 0) match {
      case (true,  false) => "Fizz"
      case (false, true)  => "Buzz"
      case (true,  true)  => "FizzBuzz"
      case (false, false) => i.toString
    }
  }
}
```

More types means more tests

A property-based test over arbitrary user configurations:

```
val arbitraryConfig: Arbitrary[Config] =
  Arbitrary { for {
    (d1, d2, w1, w2) <-
      arbitrary[(Int, Int, String, String)]
    } yield Config(d1 -> w1, d2 -> w2)
  }

def `Arbitrary pair of divisors: divisible by first` =
  arbitraryConfig { config: Config =>
    val runner = FizzBuzz.compile(config)
    val Config((d1, w1), (d2, _)) = config
    prop { i: Int =>
      (i % d1 == 0 && i % d2 != 0) ==>
      { runner(i) ==== w1 }
    }
  }
```

Our config type was too coarse

```
[info]   ! Arbitrary divisor/word pair fizzBuzzers
[error]  ArithmeticException: :
         A counter-example is 'Config((0,),(0,))':
         java.lang.ArithmeticException: / by zero
         (after 0 try)  (FizzBuzzSpec.scala:58)
```

- `0` as a divisor crashes!
- Discovery of client's underspecification.
- We talk to client: client meant only divisors within `2` and `100`.

Need to:

- Incorporate runtime validation when constructing `Config`.
- Correct our random `Config` generator.

Add runtime validation

(Quick and dirty) runtime precondition checking using Scala standard library throwing an exception (yuck).

```
val DIVISOR_MIN = 2
val DIVISOR_MAX = 100

def validatePair(pair: (Int, String)) = pair match {
  case (d, _) =>
    require(d >= DIVISOR_MIN,
      s"divisor $d must be >= $DIVISOR_MIN")
    require(d <= DIVISOR_MAX,
      s"divisor $d must be <= $DIVISOR_MAX")
}

case class Config(pair1: (Int, String),
                  pair2: (Int, String)) {
  validatePair(pair1); validatePair(pair2)
}
```

More notes on validation

- (No time to cover): in real life, prefer type-based solution such as Scalaz validation.
- Note: there are languages with more powerful type systems (dependent type system), such as Idris, that enable defining and checking more precise types (such as "integer within 2 and 100").
  - Heartbleed could have been prevented by coding in the systems language ATS.
- Do not use a weaker type system as an excuse not to write tedious validation code or tests!
  - Heartbleed could have been prevented using good validation and testing practices.

Improve random config generator

```
val arbitraryConfig: Arbitrary[Config] =
  Arbitrary {
    for {
      d1 <- choose(DIVISOR_MIN, DIVISOR_MAX)
      d2 <- choose(DIVISOR_MIN, DIVISOR_MAX)
      w1 <- arbitrary[String]
      w2 <- arbitrary[String]
    } yield Config(d1 -> w1, d2 -> w2)
  }
```

New test runs further, stills fails

Now refactor old code to `FizzBuzz.compile`:

```
val compile: Compiler = {
  case Config((d1, w1), (d2, w2)) =>
    // Precompute, hence "compiler".
    val w = w1 + w2
    // Return an Evaluator.
    { i =>
      (i % d1 == 0, i % d2 == 0) match {
        case (true,  false) => w1
        case (false, true)  => w2
        case (true,  true)  => w
        case (false, false) => i.toString
      }
    }
}
```

- Old tests now succeed.
- New test also succeeds.

Generalizing to more than two divisors

Our client wants us to support `FizzBuzzPop`:

- Specify three divisors, such as 3, 5, 7.
- Print a string combining segments of three words, such as `"Fizz"`, `"Buzz"`, `"Pop"`; or a numerical string if an integer is not a multiple of any of 3, 5, 7.
- Still get to choose the three words.
- Example: `21` should output `"FizzPop"`.

How to refactor to support `FizzBuzzPop`?

Thought-driven development

Software development is not primarily about coding, but thinking.

- Deep fact: solving a more general problem is often easier than solving the specific problem.
- There are four important numbers in the Universe:

0	emptiness
1	existence
2	other (relationship)
many	community

More features means more types (again)

Write new tests for a proposed `Defaults.fizzBuzzPopper`:

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

Add `Defaults.fizzBuzzPopper`:

```
val fizzBuzzPopperConfig: Config =
  Config(Seq(
    3 -> "Fizz", 5 -> "Buzz", 7 -> "Pop"
  ))

val fizzBuzzPopper: Evaluator =
  FizzBuzz.compile(fizzBuzzPopperConfig)
```

Test-driven type refactoring (again)

```
[error] ...Defaults.scala:29: not enough arguments for
  method apply:
  (pair1: (Int, String), pair2: (Int, String))
  com.franklinchen.FizzBuzz.Config in object Config
[error] Unspecified value parameter pair2.
[error]     Config(Seq(
[error]           ^
```

Change type `Config` to allow a sequence of pairs rather than just two:

```
case class Config(pairs: Seq[(Int, String)]) {
  pairs foreach validatePair
}
```

Note how our iterative development process promotes reuse (here, of validation logic).

Fix remaining type errors

Most significant required change reveals the unimplemented case of more than two divisors:

```scala
val compile: Compiler = {
  case Config(Seq((d1, w1), (d2, w2))) =>
    val w = w1 + w2

    { i =>
      (i % d1 == 0, i % d2 == 0) match {
        case (true,  false) => w1
        case (false, true)  => w2
        case (true,  true)  => w
        case (false, false) => i.toString
      }
    }
  case _ => // TODO handle more than 2
    { i => ??? }
}
```

General observations

- Return a sum of a subset of the configured words, if there is any divisor match.
- If there is no divisor match, return the numerical string.

More computation equals more types

- Each potential divisor (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

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

Demo time

Demo time!

Volunteers, please step up to demonstrate `FizzBuzzPop`!

Jigsaw puzzle time again

```scala
type Evaluator = Int => String
type Compiler = Config => Evaluator
type Rule = Int => String
val compile: Compiler = { case Config(pairs) =>
  // Precompute, hence "compiler".
  val rules: Seq[Rule] = pairs map buildRule
  // Return an Evaluator.
  { i =>
    val words: Seq[String] = rules map { rule => rule(i) }
    val combinedWords: String = words.mkString
    if (combinedWords.isEmpty)
      i.toString
    else
      combinedWords
  }
}
```

Test failure reflecting poor use of types

Did you see this coming?

```
nfo]   x Arbitrary pair of divisors: divisible by first
rror]  A counter-example is 'Config(List((8,), (32,)))'
(after 0 try)
rror]  A counter-example is '32405464'
(after 0 try - shrinked ('1150076488' -> '32405464'))
rror]  '32405464' is not equal to ''
(FizzBuzzSpec.scala:71)
```

Property-based testing again shows us the unexpected.

- Empty “fizz” and “buzz” words are a strange corner case.
- Unexpected ambiguity:

  Intended behavior Output a number only if it has none of the divisors.

  Actual behavior 1649349 is divisible by 13 but not 91, yet 1649349 was output.

Why?

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 another type!

`Option[A]` type

`Option[A]` is one of two possibilities:

- `None`
- `Some(a)` wraps a 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:

```
und   : String
quired: Option[String]
     word
     ^
und   : String("")
quired: Option[String]
     ""
     ^
und   : Seq[Option[String]]
quired: Seq[String]
     val words: Seq[String] = rules map { rule => rule(i) }
                                    ^
```

Fix the type errors

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

val compile: Compiler = { case Config(pairs) =>
  val rules: Seq[Rule] = pairs map buildRule

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

Monoids

"Addition" for `Option[String]`:

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

Monoid:

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

Parallelism

- All code here using map can be parallelized; Scala provides high-performance parallel collections.
- The place where reduce is used can be parallelized because of the monoid property.
- We discovered a theoretical speedup for generalized `FizzBuzz` from $O(n)$ to $O(\log n)$ (subtleties omitted).

Parallelism (code)

```
val parallelCompile: Compiler = {
  case Config(pairs) =>
    val rules = pairs.toArray.
      toPar.
      map(buildRule)

    { i: Int => rules.
      map(rule => rule(i)).
      reduce(addOption).
      getOrElse(i.toString)
    }
}
```

Conclusion

- Tests are useful.
- Types are useful.
- Tests and types work well together to drive design and program evolution!

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.