

Exploring type-directed, test-driven development

A case study using FizzBuzz

Franklin Chen

<http://franklinchen.com/>


June 7, 2014

Pittsburgh TechFest 2014

Outline

- 1 Introduction
- 2 Original FizzBuzz problem
- 3 FizzBuzz 2: allow configuration
- 4 FizzBuzz 3: FizzBuzzPop
- 5 Parallel FizzBuzz
- 6 Conclusion

Goals of this presentation

- Give a taste of a **practical** software development **process** that is:
 - ▶ **test**-driven
 - ▶ **type**-directed
- Show everything concretely:
 - ▶ build environment
 - ▶ testing frameworks
 - ▶ all the code
- Use FizzBuzz because:
 - ▶ problem: easy to understand
 - ▶ modifications: easy to understand
 - ▶ fun!
- Encourage you to explore further
 - ▶ Have you heard the news about **Swift**? 

Test-driven development (TDD)

- Think.
- Write a test that **fails**.
- Write code until test **succeeds**.
- Repeat, and **refactor** as needed.

Is TDD dead?

My answer: No.

Type systems

What is a type system?

For this presentation: a **syntactic** method for **proving** the absence of certain program behaviors.

“Debating” types “versus” tests?

Let's just use both!

Poor versus decent type systems

Poor type systems

- (Developed using 1960s-1970s knowledge)
- C, C++, Objective C
- Java

Decent type systems

- (Developed using 1980s-1990s knowledge)
- ML (**Standard ML**, **OCaml**, **F#**): I first used for work in 1995
- **Haskell**: I first used for work in 1995
- **Scala**: first released in 2004
- **Rust**: not yet version 1.0
- **Swift**: announced by Apple on June 2, 2014!

Original FizzBuzz problem

FizzBuzz defined

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



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



SBT: build tool supporting Scala, Java...

Killer features

- 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
```

Write type-directed stub

```
object Main extends App {  
  runToSeq(1, 100) foreach println  
  
  def runToSeq(start: Int, end: Int): Seq[String] = {  
    ???  
  }  
}
```

Write wanted type signature

??? is convenient for stubbing.

- In Scala standard library
- Just performs: `throw new NotImplementedError`

Write acceptance test (simplified)

*specs*²

Specs2: a fine testing framework for Scala, Java...

```
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"
  )
}
```

Test passes type check, but fails

Incremental compilation/testing kicks in:

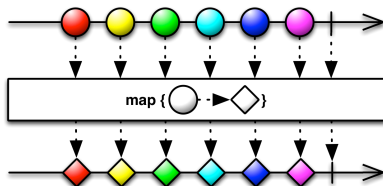
```
Waiting for source changes... (press enter to interrupt)
[info] MainSpec
[info]   x Main.runToSeq(1, 16) ==== 'strings for 1 to 16
[error]   an implementation is missing (Main.scala:19)
```

Outside-in: toward a FizzBuzz unit

Types are shapes to assemble logically.

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

- $(\text{start to end}) : \text{Seq}[\text{Int}]$, where $\text{Seq}[_]$ is a **type constructor** that, given a type A , returns a type of $\text{Seq}[A]$.
- For any value of type $\text{Seq}[A]$, $\text{map} : (A \Rightarrow B) \Rightarrow \text{Seq}[B]$.



- Therefore: need to implement function
`FizzBuzz.evaluate: Int => String`.

Start writing the FizzBuzz module

A failing acceptance test drives **discovery** of

- A **unit**, FizzBuzz
- A function with a particular type, `Int => String`

```
object FizzBuzz {  
  type Evaluator = Int => String  
  
  val evaluate: Evaluator = { i =>  
    ???  
  }  
}
```

Types are better than **comments** as **documentation**!

Comments are not checkable, unlike types and tests.

First cut at 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



ScalaCheck: a framework for writing **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" }
      }
  }
```

Killer features

- Auto-generates random tests for each property (100 by default).
- **Type-driven**: here, generates random **Int** values.

Property-based tests (continued)

The other three properties 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

```
// 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!

“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: easy to write incorrect sequences 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
  }
}
```

Killer features

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

Example of non-exhaustive pattern matching

```
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) => ???
  }
}
```

```
[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]      ^
```

Digression: Swift has pattern matching too

```
typealias Evaluator = Int -> String

let evaluate: Evaluator = { i in
    switch (i % 3 == 0, i % 5 == 0) {
    case (true,  false): return "Fizz"
    case (false, true):  return "Buzz"
    case (true,  true):  return "FizzBuzz"
    case (false, false): return String(i)
    }
}
```

Acceptance test passed, finally

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

Are we done?

No. Client wants more features.

Adding new features

Client wants to:

- Choose two **arbitrary** divisors in place of 3 and 5
 - ▶ such as 4 and 7
- Choose 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: to make new tests **type check**.
- **Refactor** original code and tests: use new APIs.
- Keep passing the **old** tests.
- Delay writing code for new features.

More features means more types

Change `Main.runToSeq` driver:

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

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

Write new 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 evaluator = FizzBuzz.compile(config)
    val Config((d1, w1), (d2, _)) = config
    prop { i: Int =>
      (i % d1 == 0 && i % d2 != 0) ==>
      { evaluator(i) ==== w1 }
    }
  }
```

Problem: coarse Config type

```
[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!**
- We discovered client's **underspecification**.
- Client says: meant to allow only divisors within 2 and 100.

We need to:

- Incorporate runtime **validation** when constructing **Config**.
- Correct our **Config** random 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)
}
```

Data validation is no joke!

- In real life, prefer non-exception type-based solution such as **Scalaz validation**.
- Also: there are languages with more powerful type systems (**dependent type systems**), 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 also have been prevented using **good validation and testing practices**.

Improve Config random 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

- But refactor old code to `FizzBuzz.compile`.
- All old tests pass; new test passes also.

```
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  
      }  
    }  
}
```

Generalizing to more than two divisors

Client wants FizzBuzzPop!

- Given three divisors (such as 3, 5, 7).
- Given three corresponding words (such as "Fizz", "Buzz", "Pop").
- Compile to evaluator that given an integer prints:
 - ▶ either a string combining a subset of the three words, or
 - ▶ a numerical string if the integer is not a multiple of any of the three divisors
- Example: 21 should output "FizzPop".

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 tests

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)
```

More tests means more (or changed) types

```
[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

Revealed: unimplemented case of more than two divisors.

```
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 means more types

- Each potential divisor (such as 3, 5, or 7 in FizzBuzzPop) should be compiled to a rule of type `Int => String`.

```
type Rule = Int => String

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


Demo time

- Given a sequence of rules and an integer: apply all the rules to the integer, then combine the partial results.

Volunteers, step up for FizzBuzzPop!

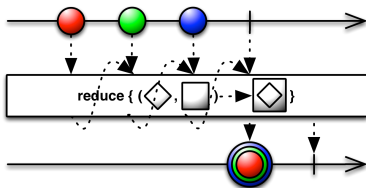
Each person will play the role of a **Rule**.

Assemble the types again

```
type Evaluator = Int => String
type Compiler = Config => Evaluator
type Rule = Int => String
val compile: Compiler = { case Config(pairs) =>
  val rules: Seq[Rule] = pairs map buildRule // compile
  // Return an Evaluator.
  { i =>
    val words: Seq[String] = rules map { rule => rule(i) }
    val combined: String = words reduce { (x, y) => x+y }
    if (combined.isEmpty)
      i.toString
    else
      combined
  }
}
```

A note on reduce

For any value of type `Seq[A]`, `reduce`: $((A, A) \Rightarrow B) \Rightarrow B$.



Example: for `Seq[String]`, reduction with string concatenation `+` returns the concatenation of all the strings in the sequence.

Test failure: coarse types again

```
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 - shrunk ('1150076488' -> '32405464'))
rror]   '32405464' is not equal to ''
(FizzBuzzSpec.scala:71)
```

Property-based testing revealed the unexpected

- 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.
- Corner case: empty “fizz” and “buzz” words.

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: refine 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.

Another Swift digression

Swift also has the option type, with special syntax.

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


Monoids

We define “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
}
```

`Option[String]` is a Monoid

- There is an identity element (`None`).
- There is a binary associative operator (`addOption`).

Parallelism

- Use of **map**: parallelizable; there are high-performance **parallel collections** for Scala.
- Use of **reduce**: parallelizable because of the monoid property.

We discovered a theoretical speedup for generalized FizzBuzz from $O(n)$ to $O(\log n)$ (omitting some technical subtleties).

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!
- Modern typed languages such as Scala promote fun, correct programming!
- It's a great time to be learning and using a modern typed language: Apple had good reasons to invent Swift.

GitHub

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.