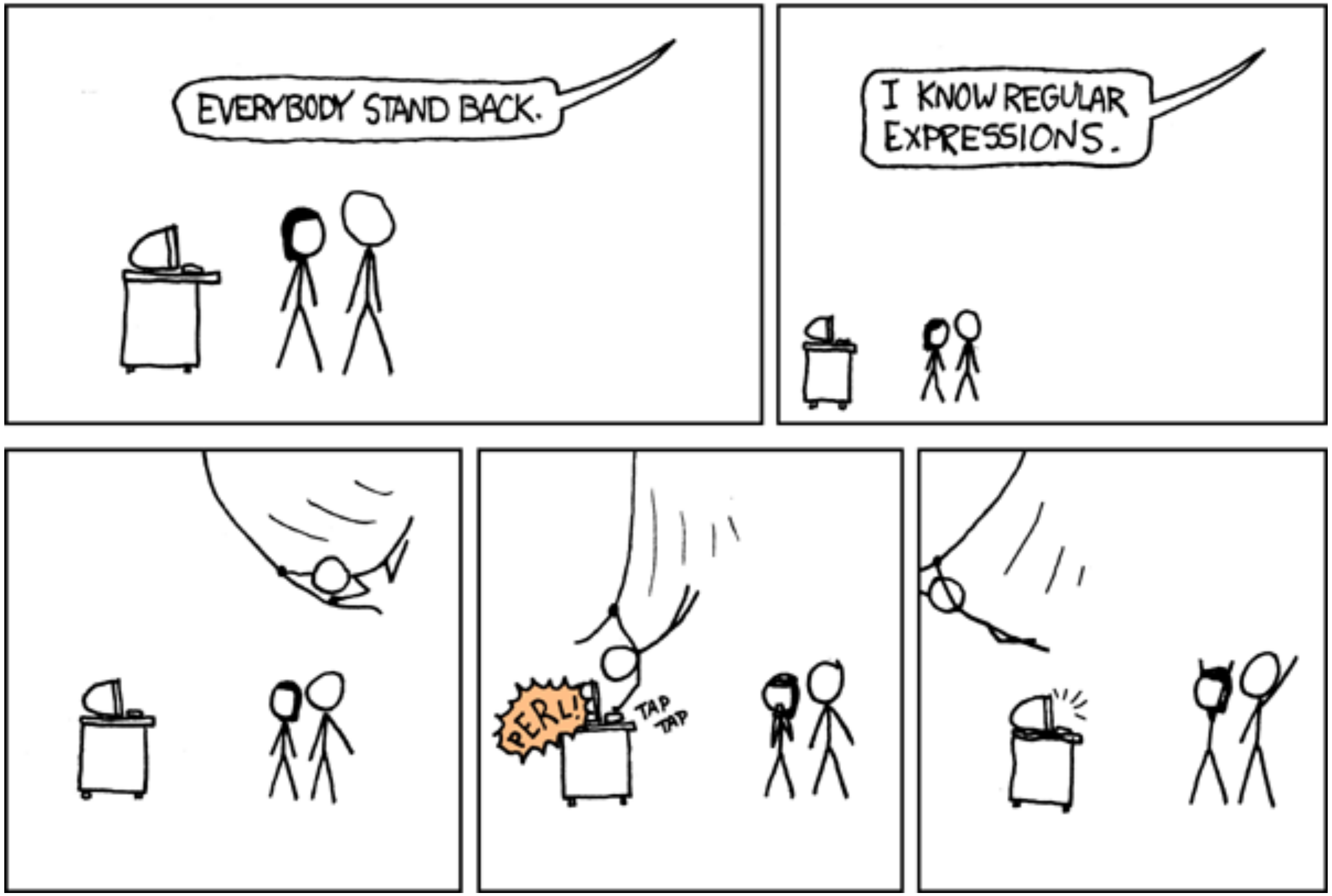
Effective Scala Programming Patterns

The most of the patterns you've got used to in a traditional OO-languages (C++, Java, C#) are just language constructs in a functional languages



The key principle behind the most
of techniques is functional languages is
composition - ability to build assets
that can be reused in different contexts
unknown at the time they created

Pimp My Library

What if you wanted to add new method for Int?

In Java you would write a wrapper class and use it **everywhere** you want your new Int on steroids.

In Scala you would use **implicit conversions**.

```
object RichInt {
implicit def intToRichInt(i: Int) = new RichInt(i)
}

class RichInt(private val value: Int) {
def minutes = value + " minutes"
}

// Usage example
import RichInt._
println(10 minutes) //println(intToRichInt(10).minutes)
```

Pimp My Library

- Simple
- Less boilerplate then ad-hoc conversion
- Can be controlled with scoping

Cell

Guess who's an evil twin

Cell is an object behaving closely like a normal variable

Implementation details are hidden from end user

Scala has two features in possesion helping us to implement such a semi-variable:

1. "First class object" functions

```
  var f: Option[() => Int] = Some(() => 1)
```

2. Syntactic sugar for update and apply

```
x() == x.apply()
```

and

```
x(<arguments>*) = y == x.update(<arguments>*, y)
```

Let's design lazy value using Cell pattern

```
class Lazy[T] { private var v: Option[T] = None private var t: Option[() => T] =
None def update(t: => T) { this.t = Some(t _) v = None } def apply() = { if(v.isEmpty)
v = Some(t.get()) v.get } } val l = new Lazy[Int] l() = { println("Evaluated"); 1} println
("Before evaluated") println(l()) l() = 2 println(l())
```

Cell actually is

- Simple once again
- Used similar to a variable thanks to syntactic sugar
- Still an object

Type class

Unified factory

```
import java.util.Date trait Vendor[T] { def vend: T } implicit object
DateVendor extends Vendor[Date] { def vend = new Date } object
Vendor { def vend[T](implicit vendor: Vendor[T]):T = vendor.vend } val
date = Vendor.vend[Date]
```

This guy is like a one man band.
Tell him a type you wanna play with and he'll hand you off an instance.

Type class

- Extendable through composition
- Abstracts capabilities from data
- Connects different class hierarchies together
- Controlable with scoping

Duck typing

SIMPLY EXPLAINED - PART 34: DUCK TYPING
geek & poke
DOCTOR, THE OTHERS SAY I DID NOT QUACK LIKE A DUCK. NOW I DON'T KNOW WHO I AM
UNTYPED DUCK

OMG, we're in J2EE environment

Following dreaded pattern can often be seen in IOC container environment(well at least when it's a bad one)

```
class DBDriver class Bean { private var driver: DBDriver = null def getDriver() = driver def setDriver(driver: DBDriver) { this.driver = driver } } class AnotherBeanFromAnotherParty { private var driver: DBDriver = null def getDriver() = driver def setDriver(driver:DBDriver) { this.driver = driver } } object Bean extends Bean { setDriver(new DBDriver) } object AnotherBeanFromAnotherParty extends AnotherBeanFromAnotherParty { setDriver(new DBDriver) }
```

Here is how we work this around with structural typing

```
def extractDriver(bean: { def getDriver(): DBDriver }): DBDriver = bean.getDriver() extractDriver (Bean) extractDriver(AnotherBeanFromAnotherParty)
```

Structural typing

- Actually a workaround for poorly designed architecture
- A typesafe workaround
- Follows don't repeat yourself

Cake
Pattern

..."Dependency Injection" is a 25-dollar term for a 5-cent concept... Dependency injection means giving an object its instance variables...

James Shore

What do you need dependency injection for?

- **reduces coupling** between components
- **simplifies configuration** of an assembly
  - useful for unit testing

IoC containers take a simple, elegant, and useful concept, and make it something you have to study for two days with a 200-page manual

Joel Spolsky

- in Scala you don't need any DI framework - forget Spring and Guice
- you make DI with **mixins & self-type annotations** and no boilerplate

Some code...

http://goo.gl/dWh2i

What's so cool in Cake pattern?

- It's not a framework, thus **any IDE supports it** out-of-box
- If a dependency is missing, compiler will immediately let you know about it

Phantom Types

The brain of an average human is used by only 10% through the lifetime

The type system is used by an average programmer by only 10% of its power

Phantom types is a simple yet powerful way to use the type system more **effectively**

2 facts about phantom types:

- never instantiated
- used only to construct other types

Let's say, we need a data type to represent the form data extracted from an HTTP request:

```
case class FormData[+T <: Data]
(data: Map[String, String])

trait Data
trait Validated extends Data
trait Unvalidated extends Data
```

Validated and **Unvalidated** are phantom types

```
trait Form {
   val readFormData: HttpServletRequest => FormData
[Unvalidated]

   val validate: FormData[Unvalidated] => FormData
[Validated]

   val printFormData: FormData[Validated] => Any
}


(readFormData andThen validate andThen printFormData)
(request)

(readFormData andThen printFormData)(request) // FAILS!!!
```

Some code…

http://goo.gl/QAwB3