1. Types 2. ??? 3. Profit!

Kristian Domagala

BFPG 24th April, 2012

Pop Quiz: What is `tags`?

```
tags = item.tags
```

1. Collection of `Tag` object?
2. Collection of `String`s?
3. Comma delimited `String`?

All of the above!

Dependent on the code path

All paths tested, but tests were setup with path assumptions

Let's write some code

```
case class User(fName:String, lName:String)
def initials(u:User):(Char,Char) =
   (u.fName.head, u.lName.head)

val bruce = User("Bruce","Wayne")
initials(bruce)
// ('B','W')

val batman = User("Batman", "")
initials(batman)
// ...
```

What went wrong?

Made an assumption about names

What happens when you assume?

[http://wins.failblog.org/2011/02/26/epic-win-photos-assumption-win/]

Maybe it works?

Maybe all the known code-paths that lead to that point have pre-validated names?

Coincidence Oriented Programming!

What happens when someone new comes on board and adds a new code path?

Let's encode an assumption

```
type Name = NonEmptyString

case class User(fName:Name, lName:Name)
```

NonEmptyString

```
sealed trait NonEmptyString {
  def value:String
  def head:Char = value.head
}

object NonEmptyString {
  def nonEmptyString(str:String):Option[NonEmptyString] =
    if (str.isEmpty)
      None
    else
      Some(new NonEmptyString { val value = str})

  def nonEmptyString(head:Char,tail:String):NonEmptyString =
    new NonEmptyString { val value = head + tail }
}
```

Let's try it again

```
type Name = NonEmptyString
case class User(fName:Name, lName:Name)

object User {
  def create(fName:String,lName:String):Option[User] =
    for {fn <- nonEmptyString(fName)
         ln <- nonEmptyString(lName)
    } yield User(fn,ln)
}

def initials(u:User):(Char,Char) =
  (u.fName.head, u.lName.head)
```

Profit!

```
val bruce:Option[User] = User.create("Bruce","Wayne")
// Some(User(...))
bruce.map(user => initials(user))
// Some(('B','W'))

val batman:Option[User] = User.create("Batman","")
// None
batman.map(user => initials(user))
// None
```

But wait, surely more code != Profit!?

How much extra code really?

- Reduced tests

- Reduced defensive programming

Just an extension

Writing a User class > using an associative array of properties to represent a user

> So why stop at "primitives" when modeling your types

Boring!

Strings are trivial; how does this help me in the Real World™?

Real World Auth & Auth

```
trait Task { def id:Id; def noteId:Id; ... }
trait Note { def id:Id; def content:String; ... }

object Database {
  def loadTask(taskId:Id):Option[Task] = {
    // ensure that user is logged in
  }
  def loadNote(noteId:Id):Option[Note] = ...
  def updateNote(noteId:Id, newContent:String) = {
    // ensure that user is logged in
    // make sure note exists
    // check that note belongs to the logged in user
  }
}
```

Types

```
trait PasswordHash {
  def matches(other:PasswordHash):Boolean
}

trait Password
object Password {
  def hash(pwd:NonEmptyString):PasswordHash = ...
}

trait SessionToken
```

???

```
sealed trait AuthenticatedUser { def user:User }

object Authentication {

  def auth(usr:User, pwd:NonEmptyString):
      Option[AuthenticatedUser] =
    if (Password.hash(pwd).matches(usr.pwdHash))
      Some(new AuthenticatedUser { val user = usr })
    else
      None

  def auth(usr:User, tok:SessionToken):
      Option[AuthenticatedUser] = ...
}
```

Authentication Profit!

```
trait Task { def id:Id; def noteId:Id; ... }
trait Note { def id:Id; def content:String; ... }

class Database(au:AuthenticatedUser) {
  def loadTask(taskId:Id):Option[Task] = {
    // can't call unless user has been authenticated
  }
  def loadNote(noteId:Id):Option[Note] = ...
  def updateNote(noteId:Id, newContent:String) = {
    // again, can't call unless user has been authenticated
    // still need to validate noteId...
  }
}
```

Authorisation Profit!

```
sealed trait Ref[A] { def id:Id }

trait Task extends Ref[Task] { def note:Ref[Note]; ...}
trait Note extends Ref[Note] { def content:String; ...}

class Database(au:AuthenticatedUser) {
  def loadTask(taskId:Id):Option[Task] = {
    ...
  }
  def loadNote(noteId:Id):Option[Note] = ...
  def updateNote(noteRef:Ref[Note], newContent:String) = {
    // can only get Ref[Note] from Database, which implies
    // the referred note belongs to the authenticated user
  }
}
```

More Profit?

Level of constraints depends on the sophistication of the type system

You can still rule-out a number of types of bugs and help focus your search when something does go wrong

Thanks!

https://github.com/dkristian/bfpg20120424

http://kristian-domagala.blogspot.com.au/2009/04/using-type-system-for-discoverability.html