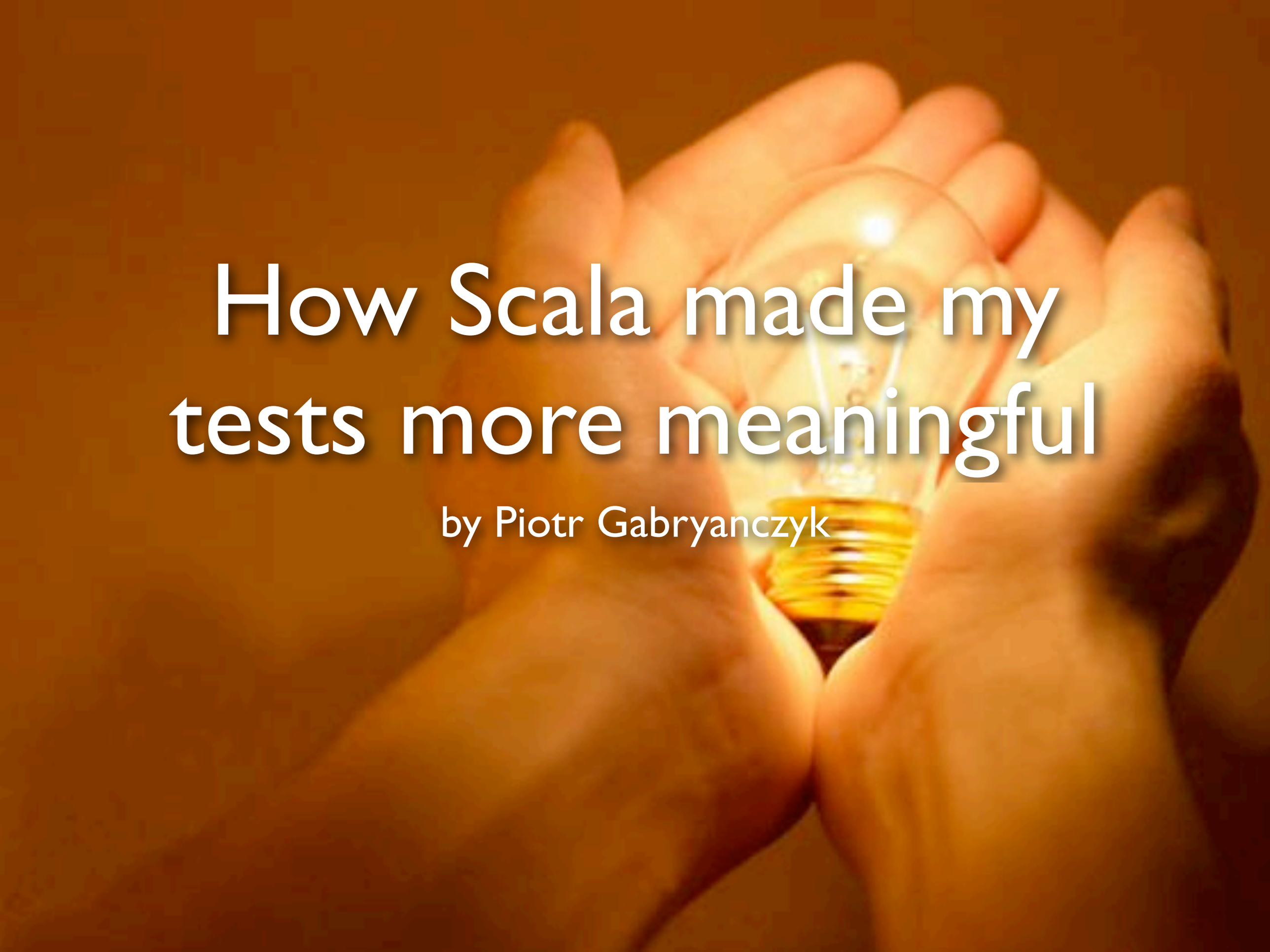


How Scala made my tests more meaningful

by Piotr Gabryanczyk

What is a good test?

- easy to write
- easy to read and understand
- easy to evolve with the system
- bang for buck
- meaningful

Which tests are meaningful?

- written using abstractions appropriate for the component you are testing
- using vocabulary defined by the API of the component

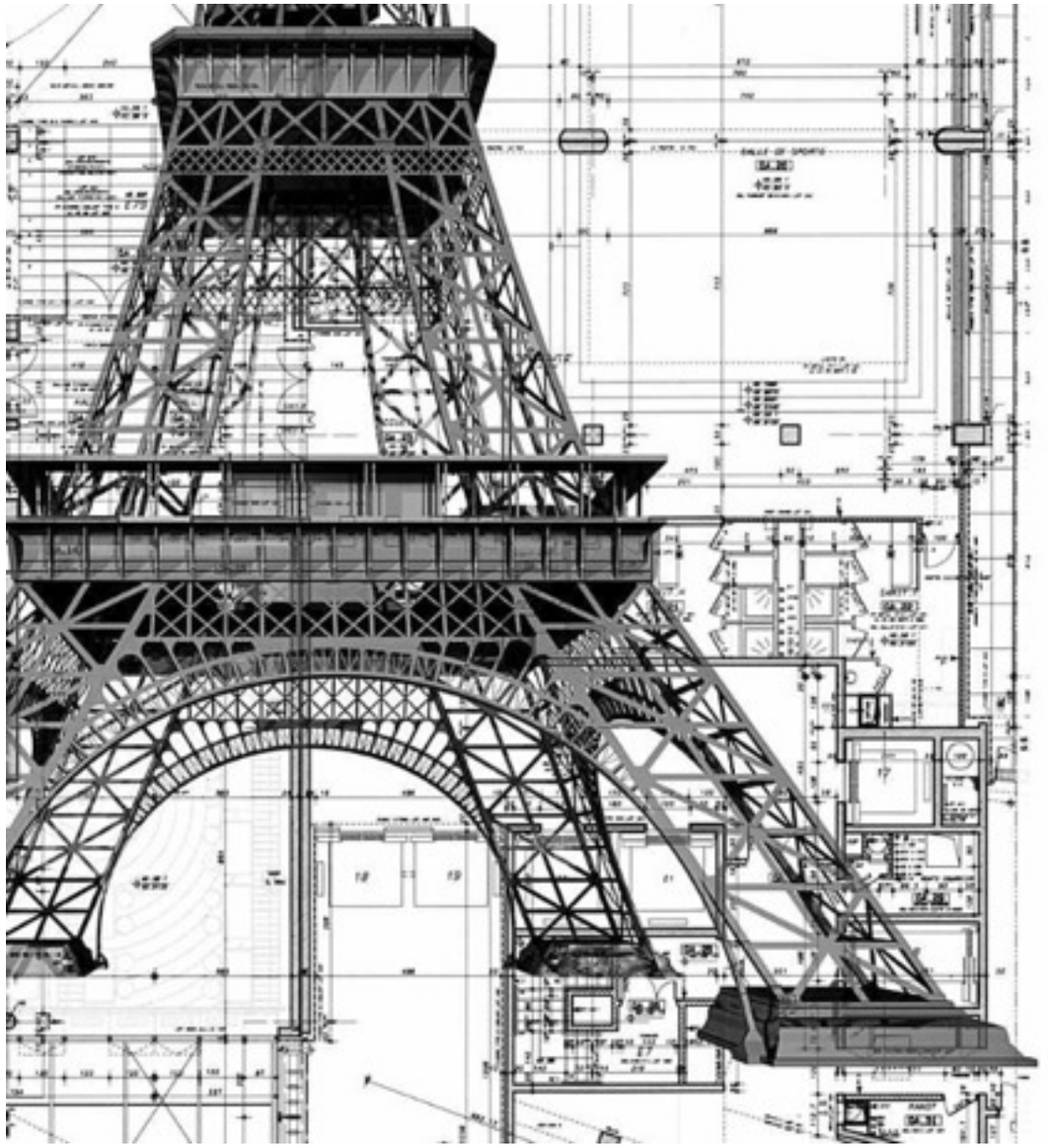
Meaningful test

```
"PersonRepository " should {

  "Save/Read" in {
    val person = TestPerson()
    repo.save(person)
    repo.read(person.id) must be (person)
  }


  "Delete" in {
    val person = TestPerson()
    repo.save(person)
    repo.delete(person.id)
    repo.read(person.id) must (beNone)
  }
}
```

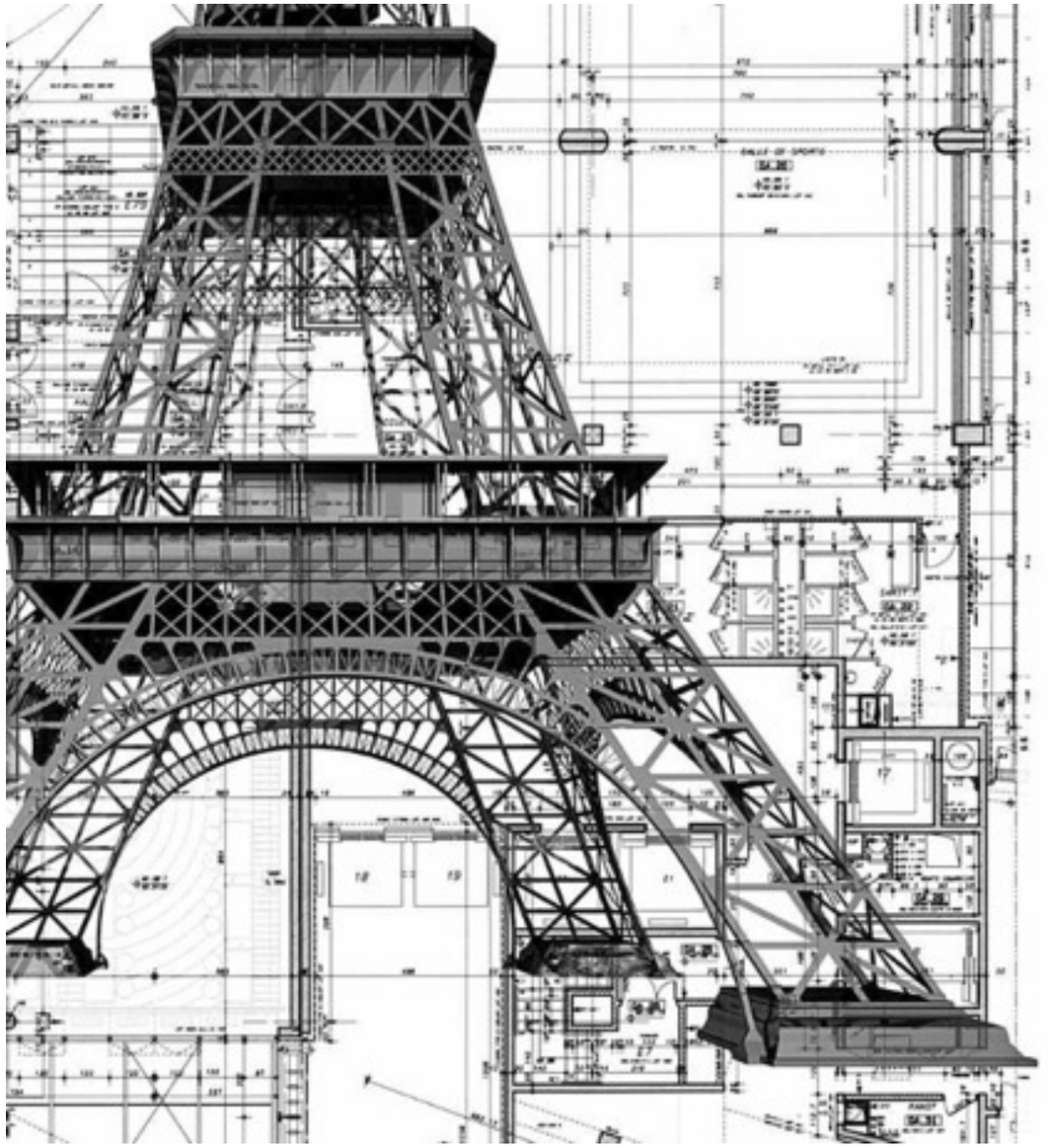
Are integration tests more meaningful?

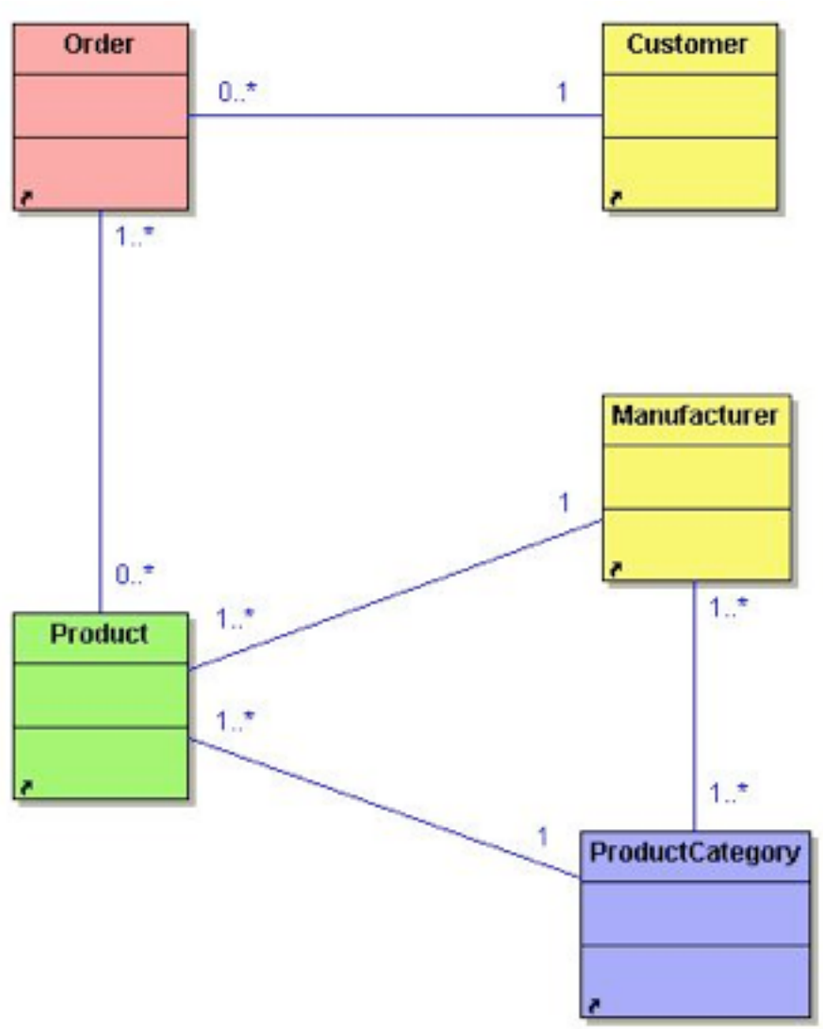
Why is it hard to write integration tests?

Why is it hard to write integration tests?

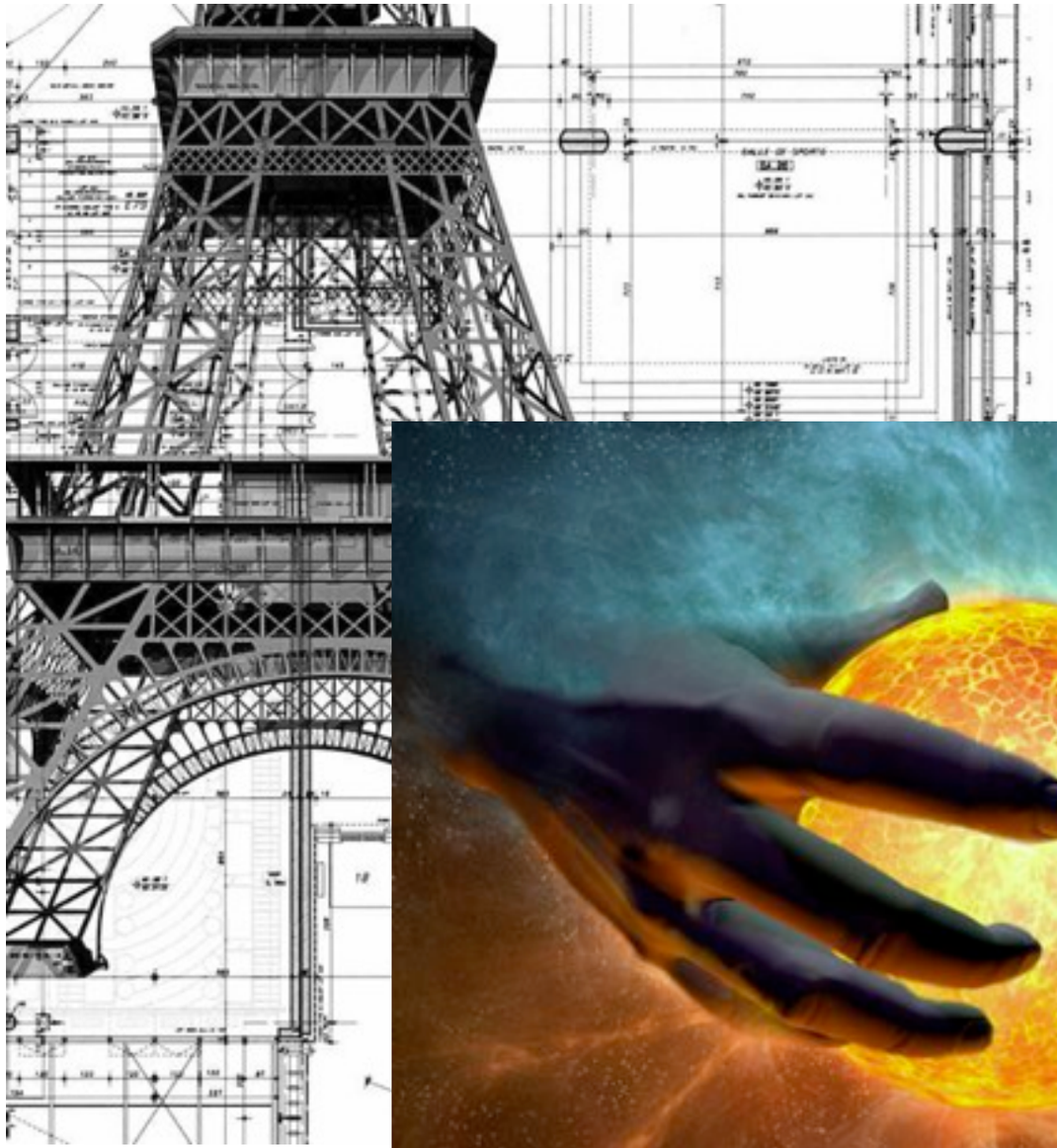


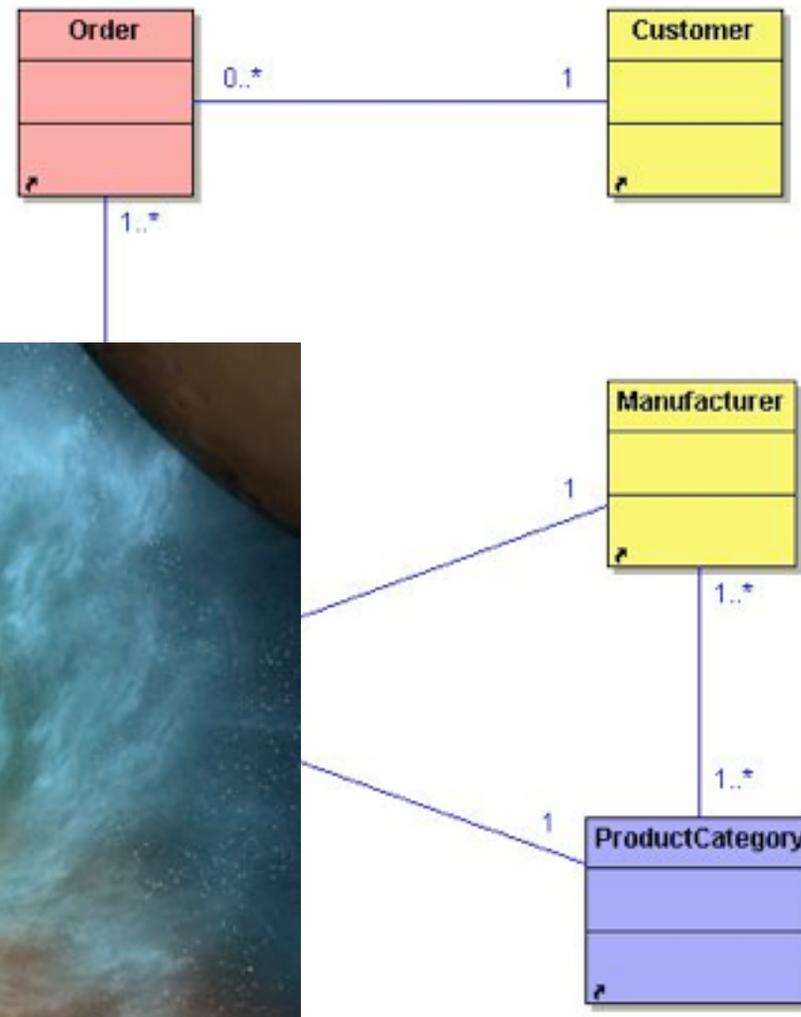
Why is it hard to write integration tests?



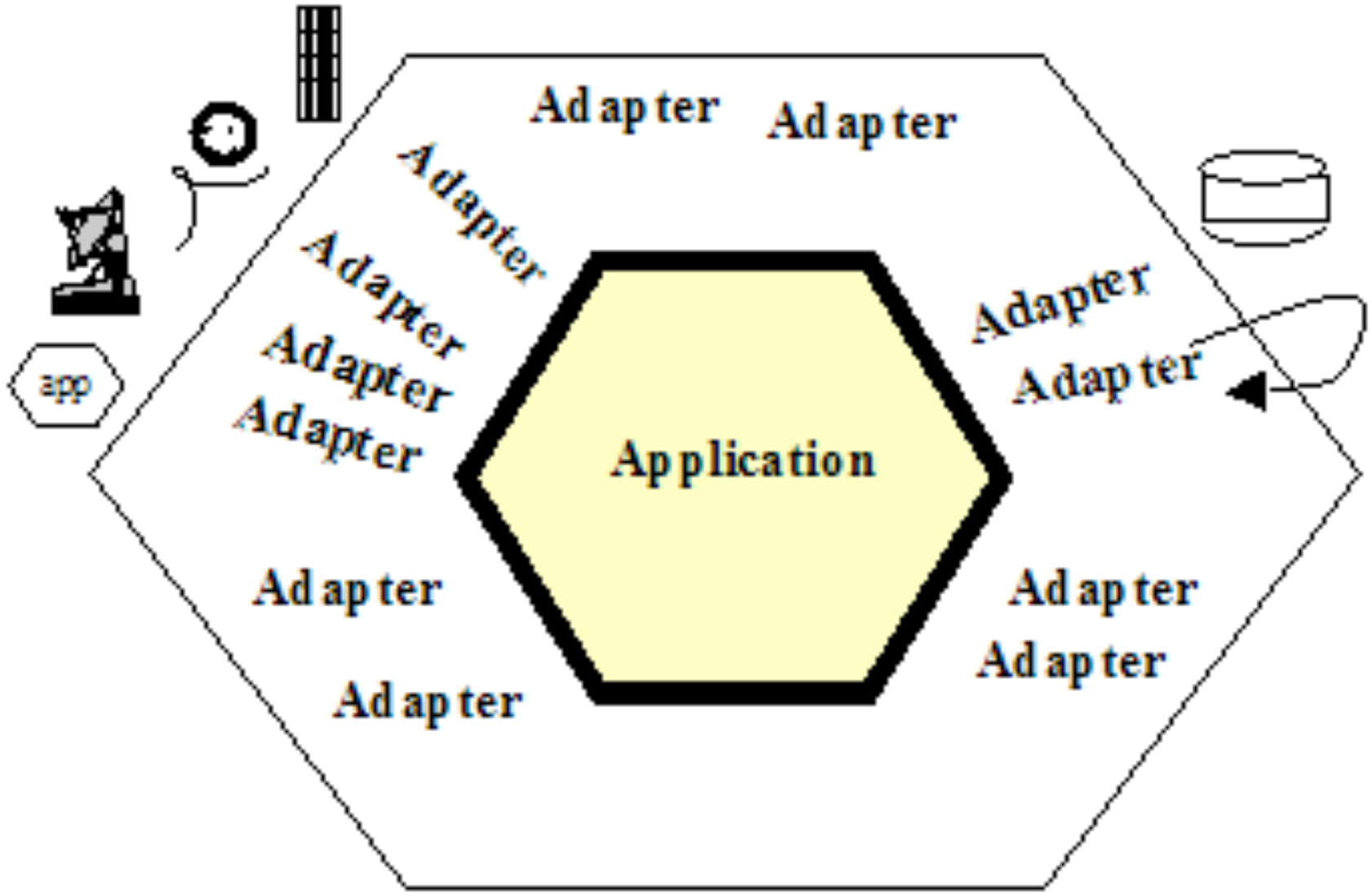


Why is it hard to write integration tests?





Hexagonal Architecture



Adapters raise the level of abstraction

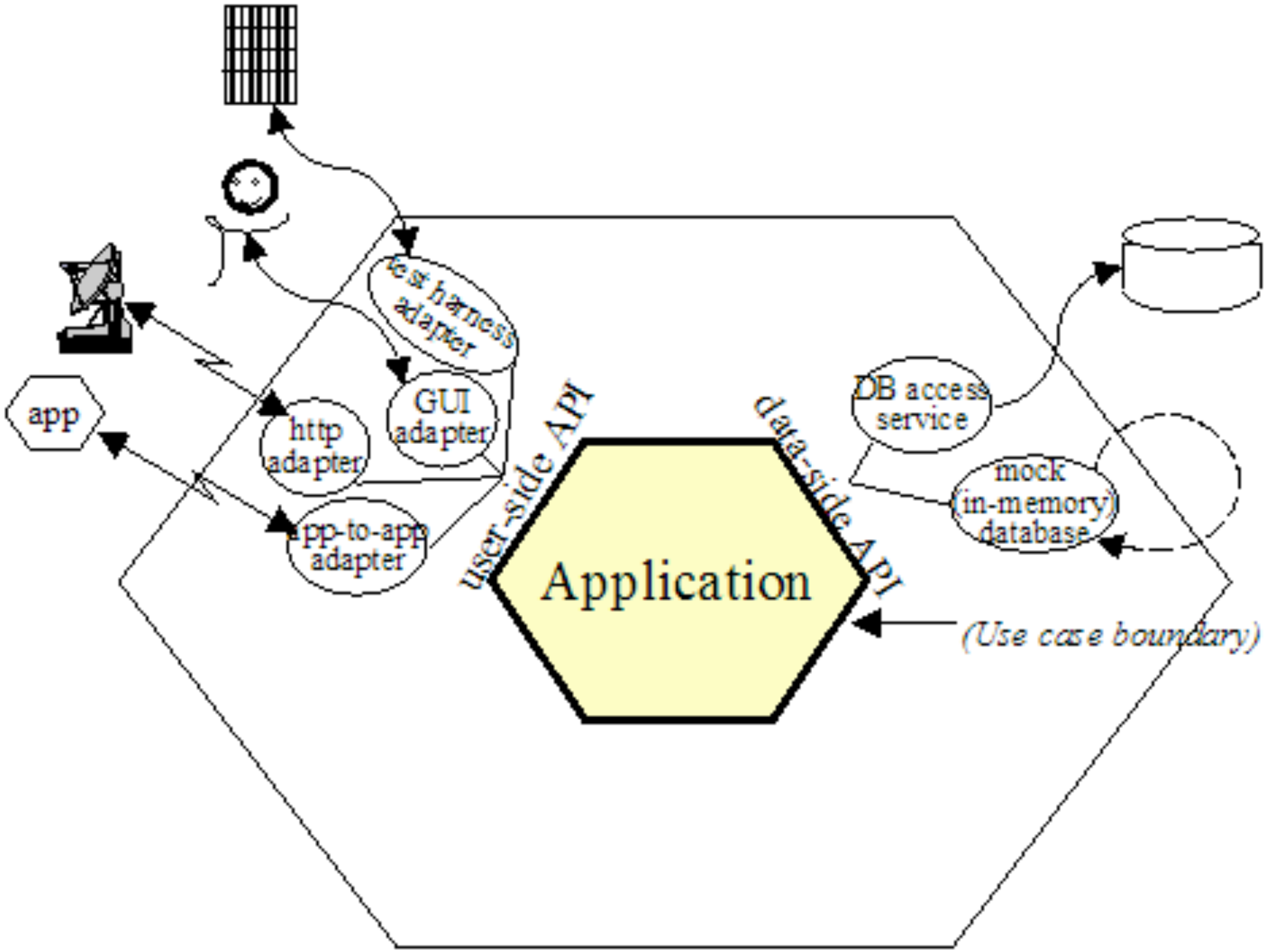
```
trait Repository{
  def save(n: KnowledgeNode)
  def findNodes(searchPhrase: String): Seq[KnowledgeNode]
  def delete(id: String)
}
```

Adapters raise the level of abstraction

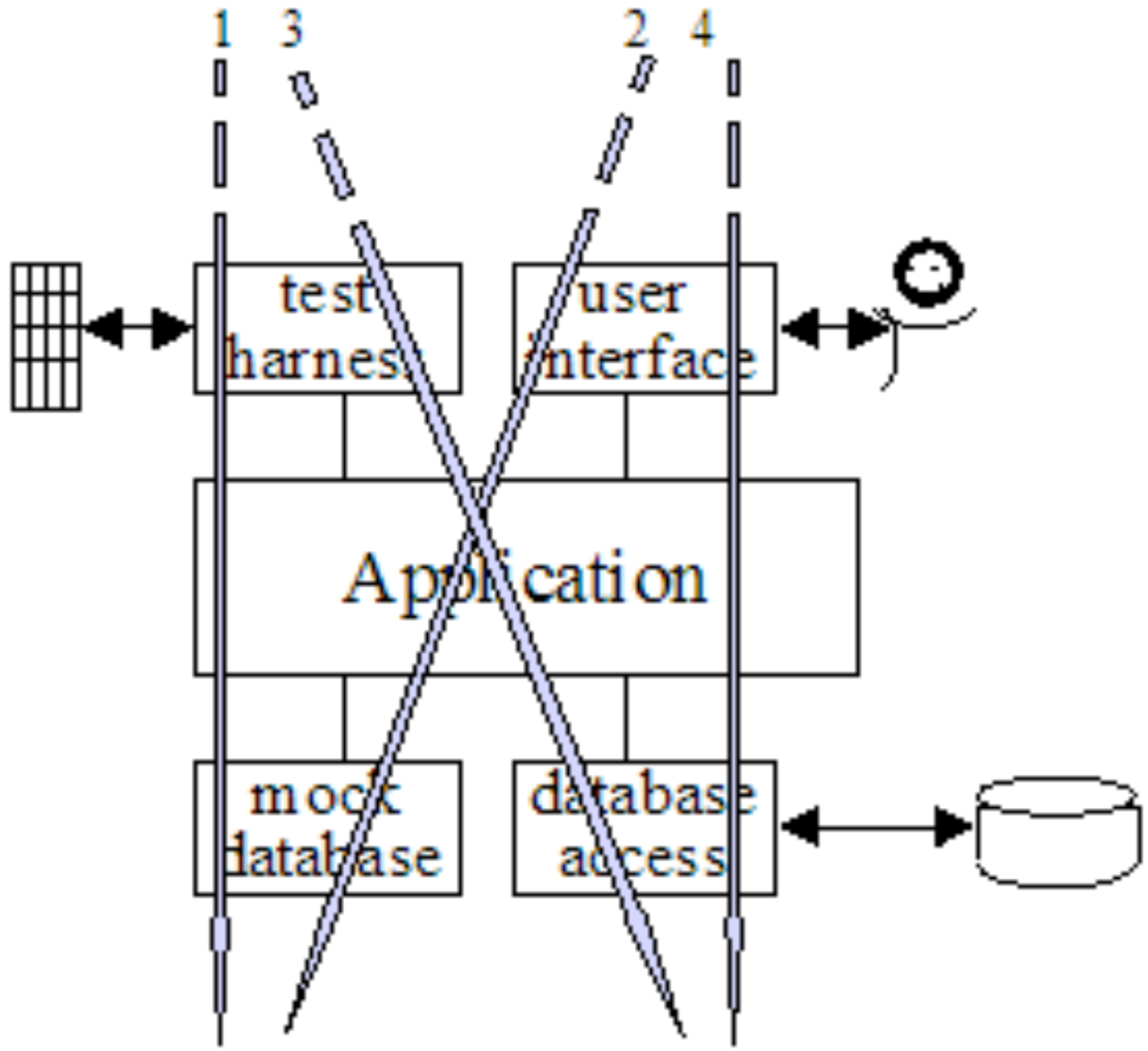
```
trait Repository{
  def save(n: KnowledgeNode)
  def findNodes(searchPhrase: String): Seq[KnowledgeNode]
  def delete(id: String)
}
```

```
trait Wikipedia{
  def findPages(keyword : String) : List[WikiId]
  def loadPageAsXml(id : WikiId) : String
}
```

Fake for every adapter



Mixing fake and real adapters



Writing fakes is easy - higher order functions

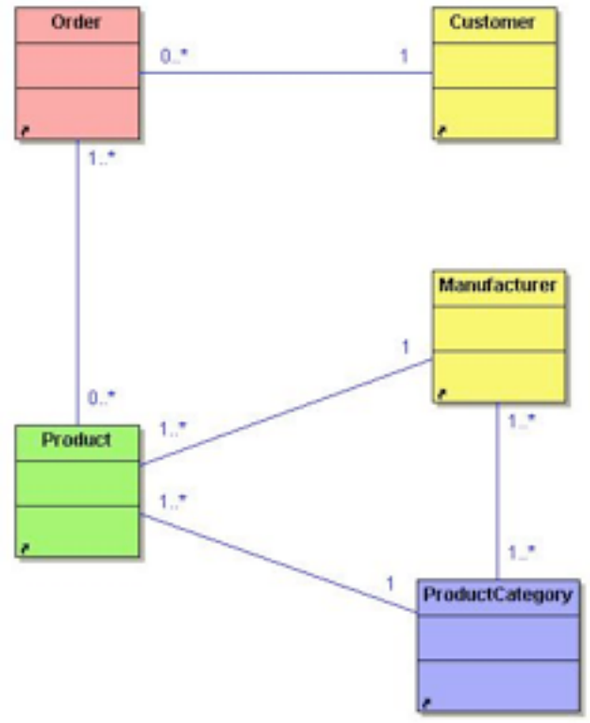
```
trait TagRepository{

  def save(tag: Tag) : Future[Unit]
  def read(tagId: String) : Future[Option[Tag]]
  def delete(TagId: String) : Future[Unit]

  def tagsForEpisode(episodeId: Int) :Future[Seq[Tag]]
  def tagsForBroadcastEvent(broadcasteventId: Int) : Future[Seq[Tag]]

  def stop()
}
```

```
class FakeTagRepository extends TagRepository{

  private val tags = new ConcurrentHashMap[String, Tag]()

  def tagsForEpisode(episodeId: Int) =
    Future{tags.values.filter(_.fields("episodeId") == episodeId.toString).toSeq}

  def tagsForBroadcastEvent(broadcasteventId: Int) =
    Future{tags.values.filter(_.fields("broadcasteventId") == broadcasteventId.toStri

  def read(tagId: String) = Future {Option(tags.get(tagId))}
  def save(tag: Tag) = Future { tags += tag.id -> tag }


  def delete(TagId: String) = Future { tags.remove(TagId) }

  def stop(){}
}
```

Very bad test data

```
lazy val category = Category(1, Some("foobar"), None)
lazy val category2 = Category(44, Some("baz"), None)
lazy val category3 = Category(44, Some("booz"), None)
lazy val enhancedWidget = WidgetAndSeq(Widget(None, "enhanced", Some("super"), 2, None, Some(1.0), None, Some(category), None, None), "2")
lazy val timelineWidgets = List(enhancedWidget)
lazy val enhancedWidget2 = WidgetAndSeq(Widget(None, "enhanced2", Some("super2"), 4, None, Some(1.0), None, Some(category2), None, None), "4")
lazy val timelineWidgets2 = List(enhancedWidget2)
lazy val enhancedWidget3 = WidgetAndSeq(Widget(None, "enhanced3", Some("super3"), 4, None, Some(1.0), None, Some(category3), None, None), "yyy")
lazy val timelineWidgets3 = List(enhancedWidget3)

lazy val categorya = Category(1, Some("Afoobar"), None)
lazy val categoryb = Category(44, Some("Bbaz"), None)
lazy val categoryc = Category(44, Some("Cbooz"), None)
lazy val enhancedWidgeta = WidgetAndSeq(Widget(None, "enhanceda", Some("super"), 2, None, Some(1.0), None, Some(categorya), None, None), "ad1")
lazy val timelineWidgetsa = List(enhancedWidgeta)
lazy val enhancedWidgetb = WidgetAndSeq(Widget(None, "enhancedb", Some("super2"), 4, None, Some(1.0), None, Some(categoryb), None, None), "ad2")
lazy val timelineWidgetsb = List(enhancedWidgetb)
lazy val enhancedWidgetc = Widget(None, "enhancedc", Some("super3"), 4, None, Some(1.0), None, Some(categoryc), None, None)
lazy val timelineWidgetsc = List(enhancedWidgetc)
```

Very bad test data




```
lazy val category = Category(1, Some("foobar"), None)
lazy val category2 = Category(44, Some("baz"), None)
lazy val category3 = Category(44, Some("booz"), None)
lazy val enhancedWidget = WidgetAndSeq(Widget(None, "enhanced", Some("super"), 2, None, Some(1.0), None, Some(category), None, None), "2")
lazy val timelineWidgets = List(enhancedWidget)
lazy val enhancedWidget2 = WidgetAndSeq(Widget(None, "enhanced2", Some("super2"), 4, None, Some(1.0), None, Some(category2), None, None), "4")
lazy val timelineWidgets2 = List(enhancedWidget2)
lazy val enhancedWidget3 = WidgetAndSeq(Widget(None, "enhanced3", Some("super3"), 4, None, Some(1.0), None, Some(category3), None, None), "yyy")
lazy val timelineWidgets3 = List(enhancedWidget3)

lazy val categorya = Category(1, Some("Afoobar"), None)
lazy val categoryb = Category(44, Some("Bbaz"), None)
lazy val categoryc = Category(44, Some("Cbooz"), None)
lazy val enhancedWidgeta = WidgetAndSeq(Widget(None, "enhanceda", Some("super"), 2, None, Some(1.0), None, Some(categorya), None, None), "ad1")
lazy val timelineWidgetsa = List(enhancedWidgeta)
lazy val enhancedWidgetb = WidgetAndSeq(Widget(None, "enhancedb", Some("super2"), 4, None, Some(1.0), None, Some(categoryb), None, None), "ad2")
lazy val timelineWidgetsb = List(enhancedWidgetb)
lazy val enhancedWidgetc = Widget(None, "enhancedc", Some("super3"), 4, None, Some(1.0), None, Some(categoryc), None, None)
lazy val timelineWidgetsc = List(enhancedWidgetc)
```

Builder Pattern

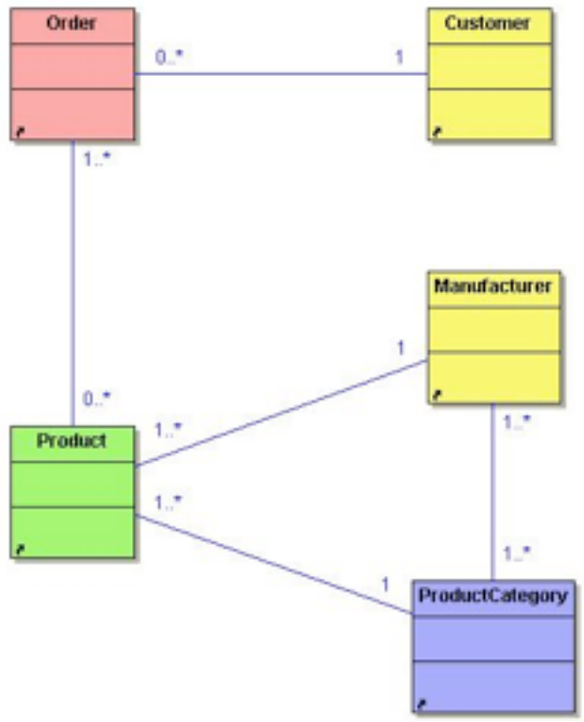
```
case class Address(street: String, postcode: String, city:String, country:String)

object TestAddress{
  def apply(street: String = "Manchester Road", postcode: String = "E14 3HG",
            city:String = "London", country:String = "UK") =
    Address(street, postcode, city, country)
}
```

Builder Pattern

```
case class Address(street: String, postcode: String, city:String, country:String)

object TestAddress{
  def apply(street: String = "Manchester Road", postcode: String = "E14 3HG",
            city:String = "London", country:String = "UK") =
    Address(street, postcode, city, country)
}
```

```
TestAddress(postcode = "KL13 AF5")
```

Builder Pattern




```
case class Address(street: String, postcode: String, city:String, country:String)

object TestAddress{
  def apply(street: String = "Manchester Road", postcode: String = "E14 3HG",
            city:String = "London", country:String = "UK") =
    Address(street, postcode, city, country)
}
```

```
TestAddress(postcode = "KL13 AF5")
```

It is easy if you build it up gradually

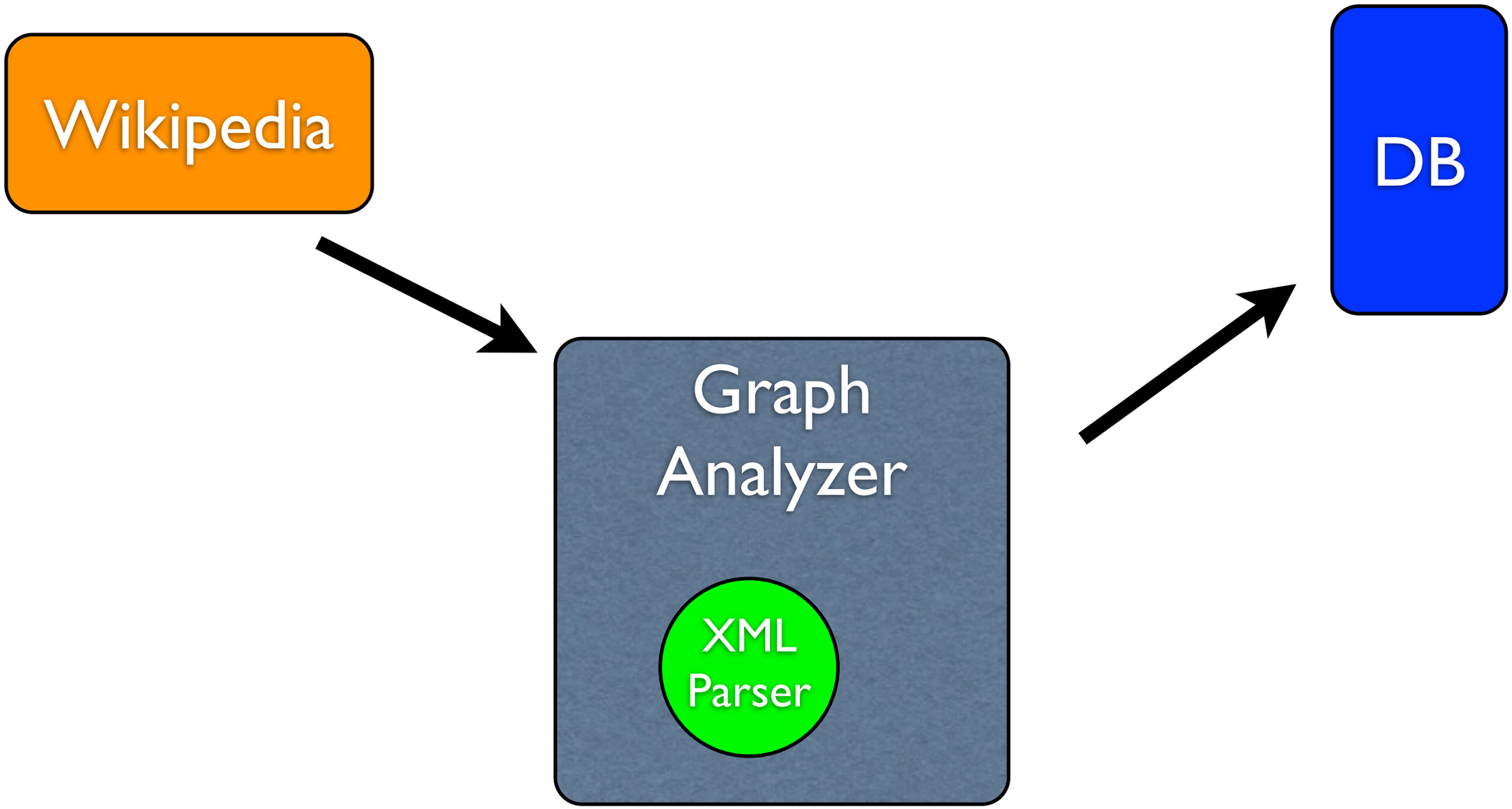
```
case class Client(name: String, address: Address, age: Short, nationality : String)

object TestClient{
  def apply(name: String= "John Smith", address: Address = TestAddress,
            age: Short = 28, nationality : String = "British") =
    Client(name, address, age, nationality)
}
```

It is easy if you build it up gradually

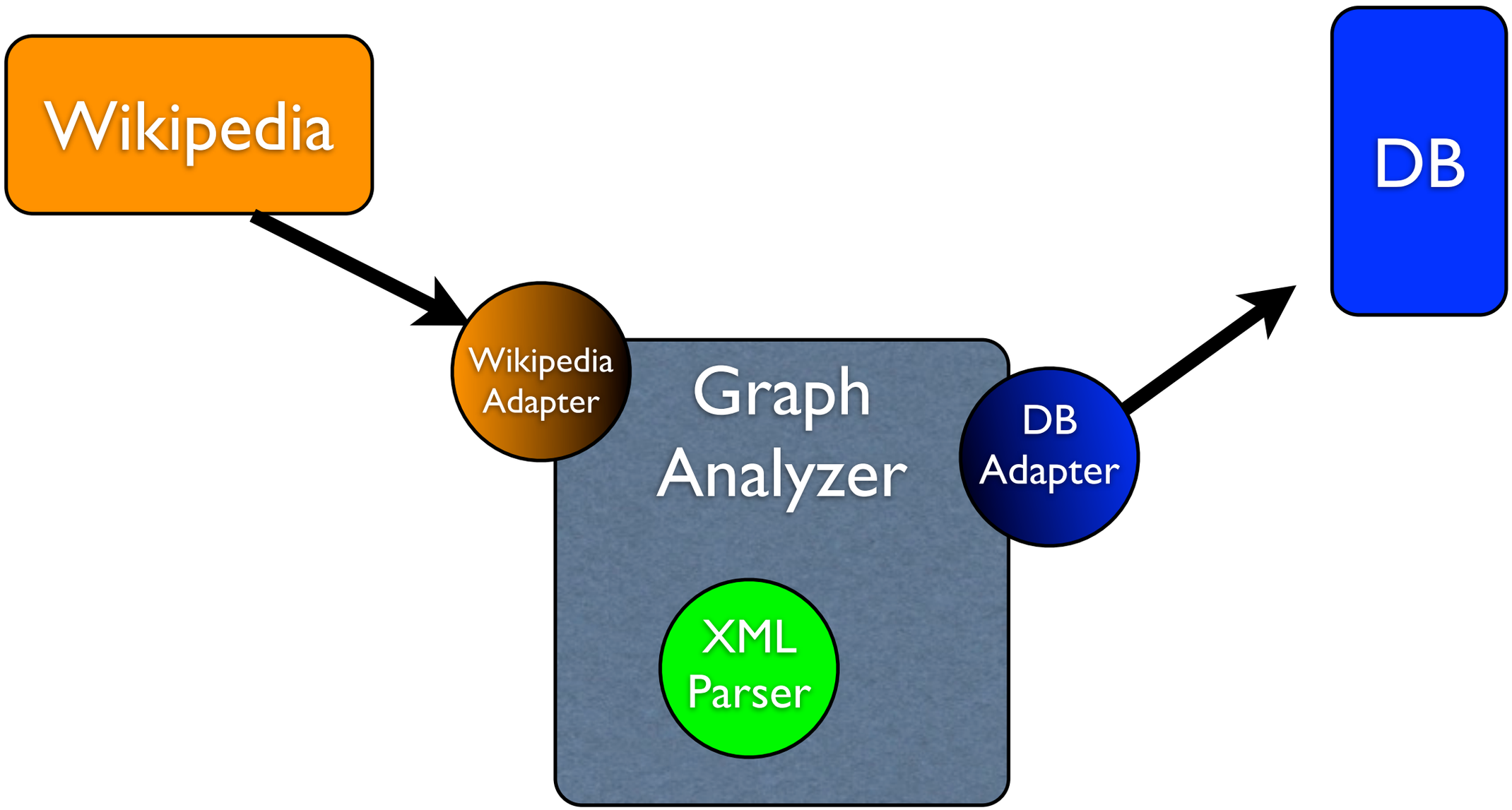
```
case class Client(name: String, address: Address, age: Short, nationality : String)

object TestClient{
  def apply(name: String= "John Smith", address: Address = TestAddress,
            age: Short = 28, nationality : String = "British") =
    Client(name, address, age, nationality)
}


case class Account(id: String, owner: Client,
                   balance: Double, premium: Boolean, interest: Double)

object TestAccount{
  def apply(id: String= UUID.randomUUID().toString,
            owner: Client = TestClient, balance: Double = 100,
            premium: Boolean = false, interest: Double = 0.04) =
    Account(id, owner, balance, premium, interest)
}
```

Everything depends on everything

Peers vs Internals



Fewer dependencies



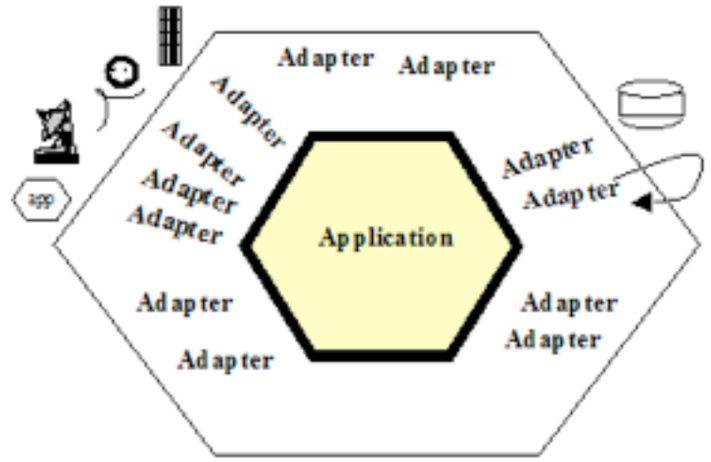
Creating sub-systems is easy!

```
val graphAnalyzer = new GraphAnalyzer(
  config.graphAnalyzer,
  FakeWikipedia(),
  RealRepo()
)
```

Creating sub-systems is easy!

```
val graphAnalyzer = new GraphAnalyzer(
  config.graphAnalyzer,
  FakeWikipedia(),
  RealRepo()
)
```

System is a one class with 5 dependencies




```
class MySystem( config: MySystemConfig,
                repo: Repository,
                facebook: Facebook,
                wikipedia : Wikipedia,
                ui: UI) {
  val graphAnalyzer = new GraphAnalyzer(config.graphAnalyzer, wikipedia, repo)
  ui.onAnalyzeGraph(keyword => graphAnalyzer(keyword))
  /*...*/
}
```

Creating of the system is easy!

```
val config = MySystemConfig.load()
```

```
val mySystem = new MySystem(
  new FakeRepository(),
  if(sys.props.contains("fakeFacebook"))
    new FakeFacebook()
  else
    new RealFacebook(config.facebook),
  new FakeTwitter(),
  new FakeUI()
)
```

Writing integration tests in Scala is Easy!

ScalaTest - FreeSpec

```scala
class SetSpec extends FreeSpec {

  "A Set" - {
    "when empty" - {
      "should have size 0" in {
        assert(Set.empty.size === 0)
      }

      "should produce NoSuchElementException when head is invoked" in {
        intercept[NoSuchElementException] {
          Set.empty.head
        }
      }
    }
  }
}
```

DSL example

```
"Twitter followers become Zeebox followers" in {
  jim follows bob onTwitter

  jim hasZeeboxAccount withTwitter
  bob hasZeeboxAccount withTwitter

  bob syncs()
  jim syncs()

  assertTrue(jim ~> bob)
  assertFalse(bob ~> jim)
}
```

Infix notation

```
class Animal{
  def eats(pray:Animal) {/*...*/}
}

val bird = new Animal
val worm = new Animal

bird eats worm
```

Implicits

```
class Animal{
  def and(other:Animal) = (this, other)
}
```

```
implicit def richPairOfAnimals(pair : (Animal, Animal)) = new {
  def areNotFriends {/*...*/}
}
```

```
bird and worm areNotFriends
```

Test Labels

```
class WikipediaTest extends FreeSpec with Labels{
  adapterTest{
    "Wikipedia adapter" - {
      "should find wikiIds for keyword House" in {
        /*...*/
      }
    }
  }
}
```

Test Labels

```
class WikipediaTest extends FreeSpec with Labels{
  adapterTest{
    "Wikipedia adapter" - {
      "should find wikiIds for keyword House" in {
        /*...*/
      }
    }
  }
}
```

```
class RepoTest extends FreeSpec with Labels{
  requiresServer("localhost:8081"){
    "Repo" - {
      "can save/read KnowledgeNode" in { /*...*/ }
    }
  }
}
```

Test Labels

```
trait Labels{

  def adapterTest(block: => Unit) {
    if(sys.props.contains("runAdapterTests")) block
  }

  def requiresServer(url:String)(block: => Unit) {
    if(serverAvailable(url)) block
  }

}
```

Matchers

```
emptySet should be ('empty)
set should not be ('empty)

result should have length (3)

map should (not be (null) and contain key ("ouch"))

string should startWith regex ("Hel*o")
string should endWith ("world")
string should include ("seven")

one should be < (7)
one should be > (0)
sevenDotOh should be (6.9 plusOrMinus 0.2)

book should have (
  'title ("Programming in Scala"),
  'author (List("Odersky", "Spoon", "Venners")),
  'pubYear (2008)
)

file should (exist and have ('name ("temp.txt")))
```

Specs2

```
class HelloWorldSpec extends Specification { def is =

  "This is a specification to check the 'Hello world' string"                ^
  p^
  "The 'Hello world' string should"                                            ^
  "contain 11 characters"                                                   ! e1^
  "start with 'Hello'"                                                      ! e2^
  "end with 'world'"                                                        ! e3^
  end

  def e1 = "Hello world" must have size(11)
  def e2 = "Hello world" must startWith("Hello")
  def e3 = "Hello world" must endWith("world")
}
```

Expecty

```
val word1 = "PING"
val word2 = "pong"

expect {
  person.say(word1, word2) == "pong pong"
}

/*
Output:

java.lang.AssertionError:

person.say(word1, word2) == "pong pong"
|      |   |      |      |
|      |   PING   pong   false
|      PING pong
Person(Fred,42)
*/
```

ScalaCheck

```
object StringSpecification extends Properties("String") {
  property("startsWith") = forAll((a: String, b: String) => (a+b).startsWith(a))

  property("concatenate") = forAll((a: String, b: String) =>
    (a+b).length > a.length && (a+b).length > b.length
  )

  property("substring") = forAll((a: String, b: String, c: String) =>
    (a+b+c).substring(a.length, a.length+b.length) == b
  )
}
```

Piotr Gabryanczyk
@piotrga
piotrga@scala-experts.com