

Easy Module Development

Cake pattern

Richard van Heest

May 2, 2017

Slides & assignments: bit.ly/dans-cake-pattern-workshop

Why cake pattern?

- Modularity
 - Usually done with packages
 - Packages are sets of files
 - Packages cannot be composed
 - Dependencies (some) are declared in **import** statements
- Traits as modules
 - ... can be composed (**extends/with**)
 - ... give typechecked dependencies
 - ... give complete encapsulation
 - ... allow to see the dependencies from the outside
 - ... is basically 'OOP the good parts'

What is the cake pattern?

- Software design pattern
- Type checked dependency injection
- No frameworks or dependencies

What is the cake pattern?

- 3 layers
 - Interface
 - Implementation
 - Wiring
- (*Almost*) everything is a trait
- **One** access point per component!

```
trait GreeterComponent {
```

```
  val greeter: Greeter
```

```
  trait Greeter {
```

```
    def greet(name: String): String = {  
      s"Hello $name!"
```

```
    }
```

```
  }
```

```
}
```

Container of what
you want to inject

Access point: the thing
you want to inject

Interface &
Implementation

What is the cake pattern?

- 3 layers
 - Interface
 - Implementation
 - Wiring
- *(Almost)* everything is a trait
- **One** access point per component!

```
trait GreeterComponent {
```

Container of what you want to inject

```
  val greeter: Greeter
```

Access point: the thing you want to inject

```
  trait Greeter {
```

```
    def greet(name: String): String = {  
      s"Hello $name!"
```

Interface & Implementation

```
    }  
  }  
}
```

Component wiring

Extends from components to be wired

```
object GreeterWiring extends GreeterComponent {
```

```
  val greeter = new Greeter {}  
}
```

Instantiates all access points

What is the cake pattern?

- 3 layers
 - Interface
 - Implementation
 - Wiring
- (*Almost*) everything is a trait
- **One** access point per component!
- Collapse interface and implementation if possible
- Use **traits** for interface/implementation as much as possible
 - Better composability
 - Better testing/mock
- Parameters instantiation in interface/implementation → use companion object for wiring

Dependencies

Inheritance

```
trait A
trait B extends A
trait C extends B
val c: C = new C {}
```

- Subtype (*'is-a'*) relationship
- B inherits all methods from A
- C inherits all methods from B and A

Self-type annotation

```
trait A
trait B { this: A => }
trait C { this: B => }
val c: C = new C with B with A {}
```


- Usage (*'required-a'*) relationship
- B can use all methods from A
- C can use all methods from B
- C **can't** use the methods from A

Dependencies

```
trait Database {  
  def query(): Any = ???  
}
```

```
trait UserDB extends Database {  
  def userData(): Any = ???  
}
```

```
trait EmailService  
  extends UserDB {  
  val userData = userData()  
  val qRes = query()  
}
```




Can call query
here

```
trait Database {  
  def query(): Any = ???  
}
```

```
trait UserDB { this: Database =>  
  def userData(): Any = ???  
}
```

```
trait EmailService {  
  this: UserDB =>  
  val userData = userData()  
  val qRes = query()  
}
```



Can't call query
here

Dependencies & inheritance

```
trait Database {  
  def query(): Any  
}  
trait SQLiteDatabase  
  extends Database {  
  def query(): Any = ???  
}  
trait MongoDBatabase  
  extends Database {  
  def query(): Any = ???  
}
```

Two implementations
of the Database

```
trait UserDB { this: Database =>  
  def getUserData(): Any = ???  
}  
trait EmailService { this: UserDB =>  
  val userData = getUserData()  
}  
  
val emailService1 = new EmailService  
  with UserDB with SQLiteDatabase {}  
val emailService2 = new EmailService  
  with UserDB with MongoDBatabase {}
```

Decide which implementation to
use while instantiating

Composing the cake

- Declare dependencies on component level
- Use the dependencies in the component

Composing the cake

```
trait GreeterComponent {  
    val greeter: Greeter  
  
    trait Greeter {  
        def greet(name: String): String = s"Hello $name!"  
    }  
}
```

Composing the cake

```
trait ConversationStarterComponent {  
  this: GreeterComponent =>
```

ConversationStarterComponent
depends on GreeterComponent

```
  val cStarter: ConversationStarter
```

```
  trait ConversationStarter {  
    def startConversation(name: String): String =  
      greeter.greet(name) + " How do you do?"  
  }  
}
```

Use the dependency
through its access point

Testing the cake

- Extend from the '*component under test*'
- Instantiate its '*access point*' with a default instance of the '*class under test*'
- In the test
 - call this default instance, or
 - create a local instance and use that in the test
- Dependencies
 - mock them (unit test)
 - instantiate them (integration test)

Testing the cake

```
class GreeterSpec extends FlatSpec with Matchers  
  with GreeterComponent {
```

Inherit from the
component

```
    override val greeter = new Greeter {}
```

Instantiate the
access point

```
    "greet" should "return a greet" in {  
      greeter.greet("Bob") shouldBe "Hello Bob!"  
    }  
  }
```

Use the instance
in your tests

Testing the cake

```
class ConversationStarterSpec extends FlatSpec with Matchers  
  with MockFactory with ConversationStarterComponent  
  with GreeterComponent {
```

Inherit from the
dependencies

Inherit from the
component

```
    override val greeter = mock[Greeter]
```

Mock the
dependencies

```
    override val cStarter = new ConversationStarter {}
```

```
    "startConversation" should "start a conversation with a greet" in {
```

```
        val name = "Bob"
```

```
        greeter.greet _ expects * once() returning s"Hello $name!"
```

Define the mock's
behavior

```
        cStarter.startConversation(name) shouldBe
```

```
            s"Hello $name! How do you do?"
```

Do the actual test

```
    }  
}
```

Assignment

Wikipedia suggestions

Assignment (Preparation)

- Check out the GitHub repository (bit.ly/dans-cake-pattern-workshop)
- Open the project in your favorite editor
- Create a package `src/main/scala/workshop5/wiki`
- Create a package `src/test/scala/workshop5/test/wiki`

Assignment

- Create a `WikipediaFacadeComponent` with a `WikipediaFacade` and its access-point
- In the `WikipediaFacade` define:
 - A value `baseUrl: String` (do not assign it here)
 - A method `search(word: String): Observable[String]` with the following implementation:

```
Observable.defer {  
    val url = baseUrl + word.replace(" ", "%20")  
  
    Observable.using(Source.fromURL(url))(r =>  
        Observable.just(r.mkString), _.close(), disposeEagerly = true)  
}
```

Assignment

- Create a `WikipediaParseComponent` with a `ResponseParser` and its access point
- In the `ResponseParser`, create an abstract function `parse(input: String): Seq[String]`

Assignment

- Create a `WikipediaParseXmlComponent` with a `XmlResponseParser` and **NO** access point
- Let them extend the `WikipediaParseComponent` and `ResponseParser`. This way you get the access point from the parent component.
- Implement the parse function:
 - `for` {
 `item <- XML.loadString(xml) \ "Section" \ "Item"`
} `yield` (`item \ "Text"`).text

Assignment

- Create a `WikipediaParseJsonComponent` with a `JsonResponseParser` and **NO** access point
- Let them extend the `WikipediaParseComponent` and `ResponseParser`.
- Implement the parse function:
 - **for** {
 `JArray(child) <- JsonMethods.parse(json)(1)`
 `JString(word) <- child`
} **yield** word

Assignment

- Create a `WikipediaSuggestionComponent` with a `WikipediaSuggestion` trait in it, as well as its access point
- Declare dependencies on the component as self-type annotations:
 - `WikipediaFacadeComponent`
 - `WikipediaParseComponent`
- In `WikipediaSuggestion` implement a method `suggestArticles(word: String): Observable[String]`
 - `wikipediaFacade.search(word)`
 `.map(responseParser.parse)`
 `.flatMapIterable(identity)`

Assignment

- Create an object Main that extends
 - App
 - WikipediaSuggestionComponent
 - WikipediaFacadeComponent
 - Any implementation of the WikipediaParseComponent
- Assign values to the three access points
 - For the baseUrl use either (depending on the implementation choice above):
 - <https://en.wikipedia.org/w/api.php?action=opensearch&format=xml&search=>
 - <https://en.wikipedia.org/w/api.php?action=opensearch&format=json&search=>
- Call and run
 - `wikipediaSuggest.suggestArticles("Hello World").subscribe(println, _.printStackTrace())`

Assignment

- Write tests for the `WikipediaSuggestionComponent` while mocking the dependencies.
- `"suggestArticles"` should `"call the wikipedia API, interpret the response and return the result"` in {
 `wikipediaFacade.search _` expects `"Hello World"` returning
 `Observable.just("some kind of result")`
 `responseParser.parse _` expects `*` returning `Seq("foo", "bar", "baz")`

```
val testSubscriber = TestSubscriber[String]()
wikipediaSuggest.suggestArticles("Hello
World").subscribe(testSubscriber)

testSubscriber.assertValues("foo", "bar", "baz")
testSubscriber.assertNoErrors()
testSubscriber.assertCompleted()
}
```