Taming the Cake Pattern with Type Macros

John Sullivan
June 12, 2013
Scala Days
Outline

• Dependency injection
  – Simple example with pseudocode
• Using the cake pattern
• Problems with the cake pattern
  – Addressing these problems with type macros
• Advantages of using the cake pattern
Outline

• Dependency injection
  – Simple example with pseudocode

• Using the cake pattern

• Problems with the cake pattern
  – Addressing these problems with type macros

• Advantages of using the cake pattern
Dependency Injection

• Decouple the process of obtaining your dependencies from the code that is using them
• Makes it easier to unit test
• More generally, makes it easier to provide different implementations of dependencies
  – Separating API from implementation
Separating Interface from Implementation

http://www.martinfowler.com/articles/injection.html
Outline

• Dependency injection
  – Simple example with pseudocode
• Using the cake pattern
• Problems with the cake pattern
  – Addressing these problems with type macros
• Advantages of using the cake pattern
case class U(uName: String)

trait URepository {
  def getU(uName: String): Option[U] = Some(U(uName))
}

class Application {
  // not dependency injection!
  val uRepository = new URepository {}
  println(uRepository.getU("uName"))
}

// outputs Some(U(uName))
Separate API from Impl

trait URepositoryApi {
  def getU(uName: String): Option[U]
}

trait URepositoryImpl extends URepositoryApi {
  def getU(uName: String): Option[U] = Some(U(uName))
}

class Application {
  // still not dependency injection
  val uRepository: URepositoryApi = new URepositoryImpl {}
  println(uRepository.getU("uName"))
}
  // outputs Some(U(uName))
Abstract API and Impl

```scala
case class U(uName: String)

trait URepository {
  def getU(uName: String): Option[U] = Some(U(uName))
}

class Application {
  // still not dependency injection
  val uRepository: api[URepository] = new impl[URepository] {}
  println(uRepository.getU("uName"))
}

// outputs Some(U(uName))
```

Make the Injection Site Abstract

trait URepository {
  def getU(uName: String): Option[U] = Some(U(uName))
}

abstract class Application {
  val uRepository: api[URepository]
  println(uRepository.getU("uName"))
}

// outputs Some(U(uName))
new Application {
  // dependency injection!
  override val uRepository = new impl[URepository] {}
}
Abstract the Concept of a Component

case class U(uName: String)

trait URepository {
  def getU(uName: String): Option[U] = Some(U(uName))
}

abstract class Application extends componentApi[URepository] {
  println(uRepository.getU("uName"))
}

// outputs Some(U(uName))
new Application with componentImpl[URepository]
DI Example – More Components

trait SRepository {
  def getS(sName: String): Option[S] = Some(S(uName))
}

trait TRepository {
  def getT(tName: String): Option[T] = Some(T(uName))
}

trait URepository {
  def getU(uName: String): Option[U] = Some(U(uName))
}
Assemble the Larger Application

abstract class Application extends
    componentApi[SRepository] with
    componentApi[TRepository] with
    componentApi[URepository] {
    println(sRepository.getS("sName"))
    println(tRepository.getT("tName"))
    println(uRepository.getU("uName"))
    }

// outputs Some(S(sName)) Some(T(tName)) Some(U(uName))
object ApplicationTest extends App {
    new Application with componentImpl[SRepository] with
        componentImpl[TRepository] with componentImpl[URepository]
}
"Extract a Container for Components"

trait Context extends 
  hasPart[SRepository] with 
  hasPart[TRepository] with 
  hasPart[URepository]

abstract class Application extends componentApi[Context] { 
  println(sRepository.getS("sName"))
  println(tRepository.getT("tName"))
  println(uRepository.getU("uName"))
}

// outputs Some(S(sName)) Some(T(tName)) Some(U(uName))
object ApplicationTest extends App {
  new Application with componentImpl[Context]
Components with Dependencies

trait URepository {
  def getU(uName: String): Option[U] = Some(U(uName))
}

trait UService extends hasDependency[URepository] {
  def getU(uName: String): Option[U] = uRepository.getU(uName)
}

trait Context extends hasPart[SRepository] with hasPart[TRepository] with hasPart[UREpository] with hasPart[SService] with hasPart[TService] with hasPart[UService]
abstract class Application extends componentApi[Context] {
    println(sService.getS("sName"))
    println(tService.getT("tName"))
    println(uService.getU("uName"))
}

// outputs Some(S(sName)) Some(T(tName))
// Some(U(uName))
object ApplicationTest extends App {
    new Application with componentImpl[Context]
}
Component Substitution

trait AltURepository extends standsInFor[URepository] {
  override def getU(name: String): Option[U] = None
}

// shortcut: take the other services from the original context
trait AltContext extends standsInFor[Context] with
  Context with
  hasPart[AltURepository]

// outputs Some(S(sName)) Some(T(tName)) None
new Application with componentImp[AltContext]
Mocking Components

class UServiceSpec extends FlatSpec with easyMock[URepository] with componentImpl[UService] {

  behavior of "UService.getU"
  it should "delegate to URepository.getU" in {
    expecting {
      uRepository.getU("kulele")
    }
    whenExecuting(uRepository) {
      uService.getU("kulele")
    }
  }
}
Outline

• Dependency injection
  – Simple example with pseudocode
• Using the cake pattern
• Problems with the cake pattern
  – Addressing these problems with type macros
• Advantages of using the cake pattern
Cake Pattern Preliminaries – Separate API from Implementation

trait URepositoryApi {
  def getU(uName: String): Option[U]
}

trait URepositoryImpl extends URepositoryApi {
  override def getU(uName: String): Option[U] = Some(U(uName))
}
Encapsulate the Type and the Injection Site

trait URepositoryComponentApi {
  val uRepository: URepositoryApi
}

trait URepositoryApi {
  def getU(uName: String): Option[U]
}

trait URepositoryComponentImpl extends URepositoryComponentApi {
  lazy val uRepository: URepositoryApi = new URepositoryImpl
  class URepositoryImpl extends URepositoryApi {
    override def getU(uName: String): Option[U] = Some(U(uName))
  }
}
trait UServiceComponentApi {
  val uService: UServiceApi
}

trait UServiceApi {
  def getU(uName: String): Option[U]
}

trait UServiceComponentImpl extends UServiceComponentApi {
  self: URepositoryComponentApi =>
  override lazy val uService: UServiceApi = new UServiceImpl
}

trait UServiceImpl extends UServiceApi {
  override def getU(uName: String): Option[U] = uRepository.getU(uName)
}
trait ContextComponentApi extends URepositoryComponentApi with UServiceComponentApi

trait ContextComponentImpl extends ContextComponentApi with URepositoryComponentImpl with UServiceComponentImpl
abstract class Application extends ContextComponentApi {
    println(uService.getU("uName"))
}

// outputs Some(U(uName))
object ApplicationTest extends App {
    new Application with ContextComponentImpl
}
Component Substitution

trait AltURepositoryComponentImpl
extends URepositoryComponentApi {
  lazy val uRepository: URepositoryApi = new AltURepositoryImpl
class AltURepositoryImpl extends URepositoryApi {
  override def getU(uName: String): Option[U] = None
}
}

// shortcut: take the other services from the original context

trait AltContextComponentImpl extends ContextComponentImpl with
  AltURepositoryComponentImpl

// outputs None
new Application with AltContextComponentImpl
Mocking Components

class URepositoryComponentMock extends URepositoryComponent {
    override lazy val uRepository = mock[URepository]
}

class UServiceSpec extends FlatSpec with
    URepositoryComponentMock with UServiceComponentImpl {
    behavior of "UService.getU"
    it should "delegate to URepository.getU" in {
        expecting {
            uRepository.getU("kulele")
        }
        whenExecuting(uRepository) {
            uService.getU("kulele")
        }
    }
}
Outline

• Dependency injection
  – Simple example with pseudocode

• Using the cake pattern

• Problems with the cake pattern
  – Addressing these problems with type macros

• Advantages of using the cake pattern
Problems with the Cake Pattern

1. **Verbose**: Lots of boilerplate.
2. **Opaque**: hard to understand because the language structures used do not signify the user's intent.
3. **Aloof**: Compiler error messages are potentially confusing and misleading, because they address the language structures used, and not the user's intent.
trait URepositoryComponent {
  val uRepository: URepository
}

trait URepository {
  def getU(uName: String): Option[U]
}

trait URepositoryComponentImpl extends URepositoryComponent {
  lazy val uRepository: URepository = new URepositoryImpl
}

class URepositoryImpl extends URepository {
  override def getU(uName: String): Option[U] = Some(U(uName))
}
/** Gratuitous scaladoc comment here */

trait URepositoryComponent {

/** Gratuitous scaladoc comment here */
val uRepository: URepository

/** Possibly useful scaladoc comment here */
trait URepository {

/** Gratuitous scaladoc comment here */
def getU(uName: String): Option[U]
}
}

/** Gratuitous scaladoc comment here */

trait URepositoryComponentImpl extends URepositoryComponent {

/** Gratuitous scaladoc comment here */
lazy val uRepository: URepository = new URepositoryImpl

/** Gratuitous scaladoc comment here */
class URepositoryImpl extends URepository {

/** Possibly useful scaladoc comment here */
Outline

• Dependency injection
  – Simple example with pseudocode

• Using the cake pattern

• Problems with the cake pattern
  – Addressing these problems with type macros

• Advantages of using the cake pattern
Implement Psuedocode with Type Macros

• Type macros are functions that run at compile time to create new types
• Type macros can have parameters, including type parameters
• So we can inspect the underlying type, e.g., URepository, to generate the new type, e.g., api[URepository]
Taming with Type Macros

1. api[A]
2. impl[A]
3. componentApi[A]
4. componentImpl[A]
5. hasPart[A]
6. hasDependency[A]
7. standsInFor[A]
8. mock[A] (mockMethod: [B]() => B)
api\[A\] and impl\[A\]

trait URepository {
    def getU(uName: String): Option[U] = Some(U(uName))
}

// psuedocode showing how api\[A\] and impl\[A\] expand:

trait api[URepository] {
    def getU(uName: String): Option[U]
}

trait impl[URepository] extends api[URepository] with URepository
componentApi[A] and componentImpl[A]

trait componentApi[URepository] {
  val uRepository: api[URepository]
}

trait componentImpl[URepository] extends componentApi[URepository] {
  lazy val uRepository: api[URepository] =
    new impl[URepository] {}
Taming with Type Macros

1. api[A]
2. impl[A]
3. componentApi[A]
4. componentImpl[A]
5. hasPart[A]
6. hasDependency[A]
7. standsInFor[A]
8. mock[A] (mockMethod: [B]() => B)
Solving Problems

• We've removed the boilerplate
• We are now signifying our intent
• In principle, the type macros could preempt standard error messages and produce more meaningful errors
Proof of Concept: Congeal

• https://github.com/sullivan-/congeal

• Initial implementations for most type macros
  – mocks not yet fully implemented

• Currently 7 examples and 31 test cases

• Uses experimental Scala version "Kepler"
  – scalaVersion := "2.11.0-SNAPSHOT"
  – scalaOrganization := "org.scala-lang.macro-paradise"
Congeal: Current Limitations on Underlying Types

• All macros require type A to be "simple", i.e., meet the following conditions:

  1. is a trait
  2. is static (i.e., not a member of a method or trait. only objects all the way up.)
  3. no non-private[this] inner classes
  4. no members that have params or return types that derive from A
Congeal: Work in Progress

• Relax constraints on underlying types
• Basic documentation
• Make existing macros more robust
  – Expand examples and test cases
• Friendlier error messages
Outline

• Dependency injection
  – Simple example with pseudocode

• Using the cake pattern

• Problems with the cake pattern
  – Addressing these problems with type macros

• Advantages of using the cake pattern
Advantages of Using the Cake Pattern

• Type-safe!
  – Most Java-style dependency injection solutions depend on run-time configuration
    • Leads to errors at run-time
    • Hard to write tests for production configuration
      – Production config may depend on external resources that are not available in test environment

• Underlying type constructs are very flexible
  – This allows us to create components that are more expressive
trait Repository extends 
  hasPart[SRepository] with 
  hasPart[TRepository] with 
  hasPart[URepository]

trait Service 
  hasDependency[Repository] with 
  hasPart[SService] with 
  hasPart[TService] with 
  hasPart[UService]

trait Root extends 
  hasPart[Repository] with 
  hasPart[Service]
Confusing Error Message

• Should be able to provide a nicer error message out of the type macro...

[error] Root.scala:6: illegal inheritance;
[error] self-type ServiceComponentImpl does not conform to UServiceComponentImpl's selftype UServiceComponentImpl with URepositoryComponentApi
[error] with UServiceComponentImpl
[error] ^
Better Error Message

• Should be able to provide a nicer error message out of the type macro...

```
[error] Root.scala:6: unsatisfied dependency;
[error] Service must declare URepository as a part or a dependency, since UService declares a dependency on URepository
[error] with hasPart[UService]
[error] ^
```
Application Design Constraints

Example Web Application Component Model:
- Controller Component
- Service Component
- Repository Component
Application Design Constraints

Example GUI Desktop Application Component Model:

UI Component

Model Component

View Component

Controller Component

Service Component
trait LetterFactory extends hasDependency[SFactory] with hasDependency[TFactory] with hasDependency[UFactory] {

  def createLetter(letter: Char, name: String) = {
    letter match {
      case 'S' => sFactory.createLetter(name)
      case 'T' => tFactory.createLetter(name)
      case 'U' => uFactory.createLetter(name)
      case _ => throw new Error(s"bad letter $letter")
    }
  }
}
Hidden Dependencies

trait Factory extends
  hasPart[LetterFactory] with
  hasPrivatePart[SFactory] with
  hasPrivatePart[TFactory] with
  hasPrivatePart[UFactory]

trait componentApi[Factory]
  extends componentApi[LetterFactory]

trait componentImpl[Factory] extends
  componentApi[Factory] with
  componentImpl[LetterFactory] with
  componentImpl[SFactory] with
  componentImpl[TFactory] with
  componentImpl[UFactory]
One Impl for Two Apis

```scala
trait UiReadOnlyModel {
  def getUOption: Option[U]
}

trait UiWriteOnlyModel {
  def setUOption(uOption: Option[U]): Unit
}

trait UiModel extends standsInFor[UiReadOnlyModel] with standsInFor[UiWriteOnlyModel] {
  private var uOption: Option[U] = None
  override def getUOption = uOption
  override def setUOption(uOption: Option[U]) {
    this.uOption = uOption
  }
}
```
trait UiView extends hasDependency[UiReadOnlyModel] {
  def init: Unit = println("hi from initView")
}

trait UiController extends
  hasDependency[UiWriteOnlyModel] with
  hasDependency[UiView] {
  def init: Unit = println("hi from initController")
}

trait Ui extends
  hasPart[UiModel] with
  hasPart[UiView] with
  hasPart[UiController]
trait Application extends componentApi[Ui] {
    println(uiReadOnlyModel.getUOption)
    uiWriteOnlyModel.setUOption(Some(U("uName")))
    println(uiReadOnlyModel.getUOption)
}

// outputs None Some(U(uName))
object ApplicationTest extends App {
    new Application with componentImpl[Ui]
}
Thanks for Coming!

http://scabl.blogspot.com/p/cbdi.html
https://github.com/sullivan-/congeal
john.sullivan.mscs@gmail.com
Extra Slides
abstract class Application extends componentApi[URepository] {
    println(uRepository.getU("uName"))
}

abstract class Application extends hasDependency[URepository] {
    println(uRepository.getU("uName"))
}

// outputs Some(U(uName))
new Application with componentImpl[URepository]
Why Repeat the Type here?

trait URepositoryComponentImpl extends URepositoryComponentApi {

  lazy val uRepository: URepositoryApi = new URepositoryImpl

private class URepositoryImpl extends URepositoryApi {
    override def getU(uName: String): Option[U] = None
  }
}

• Without explicitly typing uRepository, type inference will give it type URepositoryImpl
• If you make URepositoryImpl a private inner class, this becomes a problem
hasPart[A]

• hasPart[A] expands differently depending on context
• Outside of any components, hasPart[A] expands to nothing
  – (At least, nothing that matters to the user)

trait Context extends
  hasPart[URepository] with
  hasPart[UService]

// expands to:

trait Context
hasPart[A]

• Inside a component, hasPart[A] expands to a component

```scala
trait componentApi[Context] extends 
  componentApi[URepository] with 
  componentApi[UService]

trait componentImpl[Context] extends 
  componentApi[Context] with 
  componentImpl[URepository] with 
  componentImpl[UService]
```
hasDependency[A]

- hasDependency[A] expands differently depending on context
- Outside of any components, hasDependency[A] expands to componentApi[A]

```scala
trait UService extends hasDependency[URepository] {
  def getU(uName: String): Option[U] =
    uRepository.getU(uName)
}
```

// expands to:

```scala
trait UService extends componentApi[URepository] {
  def getU(uName: String): Option[U] =
    uRepository.getU(uName)
}
```
hasDependency[A]

• Inside componentImpl, hasDependency[A] expands to self-type

trait componentApi[UService] {
  val uService: api[UService]
}

trait componentImpl[UService] extends componentApi[UService] {
  self =>
  componentApi[URepository]

    // needed because UService extends componentApi[URepo]:
    lazy val uRepository = self.uRepository
  }
}
standsInFor[A]

trait AltURepository extends standsInFor[URepository] {
  def getU(uName: String): Option[U] = None
}

// expands to:

trait AltURepository extends api[URepository] {
  def getU(uName: String): Option[U] = None
}
standsInFor[A]

// inside a component, standsInFor expands like so:

trait componentApi[AltURepository] extends componentApi[URepository]

trait componentImpl[AltURepository] extends componentApi[AltURepository] {
  lazy val uRepository: api[URepository] = new impl[AltURepository] {}
}
Why Inner Classes Are Hard

trait DummyProvider {
  class Dummy(val i: Int)
  def provideDummy(i: Int): Dummy = new Dummy(i)
}

// expansions:
trait api[DummyProvider] {
  trait api[Dummy] { val i: Int }
  def provideDummy(i: Int): api[Dummy]
}

trait impl[DummyProvider] extends api[DummyProvider]
  with DummyProvider // <-- this doesn't work any more!
// i need to rewrite provideDummy method to return an
// impl[Dummy]
Why Inner Classes Are Hard

trait DummyProvider {
    class Dummy(val i: Int)
    def provideDummy(i: Int): Dummy = new Dummy(i)
}

trait api[DummyProvider] {
    trait api[Dummy] { val i: Int }
    def provideDummy(i: Int): api[Dummy]
}

trait impl[DummyProvider] extends DummyProvider with api[DummyProvider] {
    class impl[Dummy](i: Int) extends Dummy(i) with api[Dummy]
    def provideDummy(i: Int): api[Dummy] =
        // need to transform original method body to do this:
        new impl[Dummy](i: Int)
Two Views of a Single Component

trait ExposedTypes extends hasDependency[ExposedSymbols]

trait InternalTypes extends ExposedTypes with
    hasDependency[InternalSymbols]

trait ExposedSymbols extends hasDependency[ExposedTypes]

trait InternalSymbols extends ExposedSymbols with
    hasDependency[InternalTypes]

trait Reflection extends
    hasPart[ExposedTypes] with
    hasPart[ExposedSymbols]

trait Compiler extends Reflection with
    hasPart[InternalTypes] with
    hasPart[InternalSymbols]