The Cake Pattern in Practice

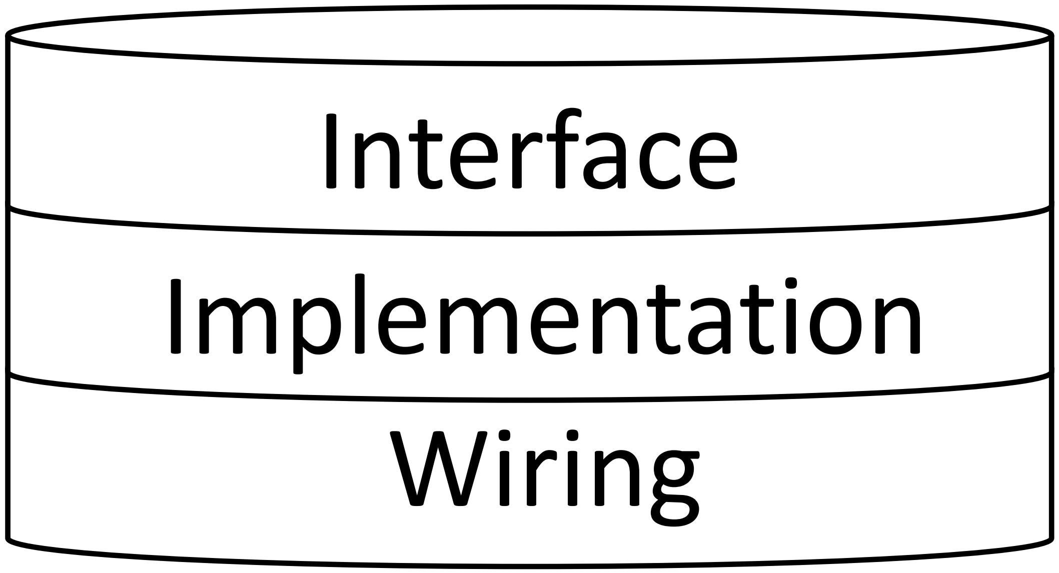
Author: Peter Potts

Date: October 8th, 2014

What is the Cake Pattern?

- Software Design Pattern
- Dependency Injection (DI)
- Aspect-Oriented Programming (AOP)
- No dependencies
- Type-Safe all the way
- First explained by Martin Odersky
- Article by Jonas Bonér

Layered Cake



Component Interface

```
trait VehicleComponent {

    val vehicle: Vehicle

    trait Vehicle
}
```

One Access Point per Component

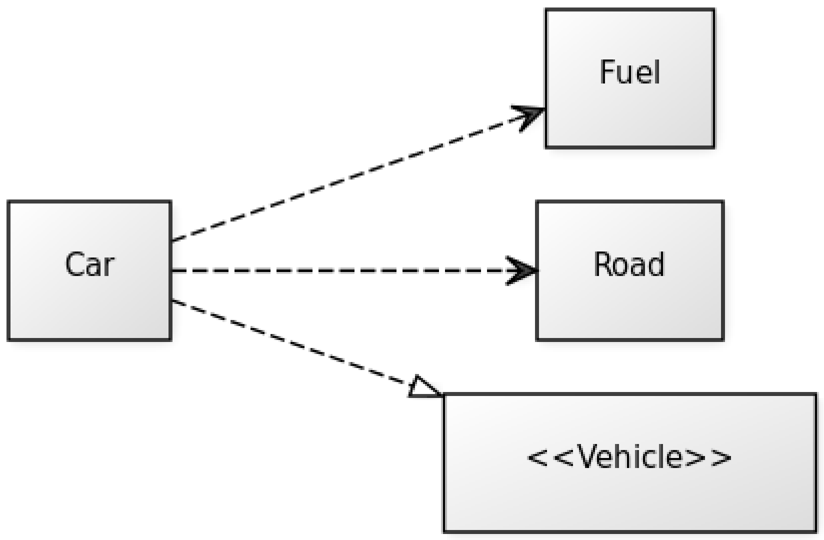
```
~~trait VehicleComponent {~~
~~    val capacity: Capacity~~
~~    val shape: Shape~~
~~}~~
```

```
trait VehicleComponent {
    val vehicle: Vehicle

    trait Vehicle {
        val capacity: Capacity
        val shape: Shape
    }
}
```

Component Implementation

```
object CarComponent {
   type Dependencies = FuelComponent with RoadComponent
}
```

```
trait CarComponent extends VehicleComponent {
   self: CarComponent.Dependencies =>

   class Car extends Vehicle {
      fuel.##
      road.##
   }
}
```



Single Component Wiring

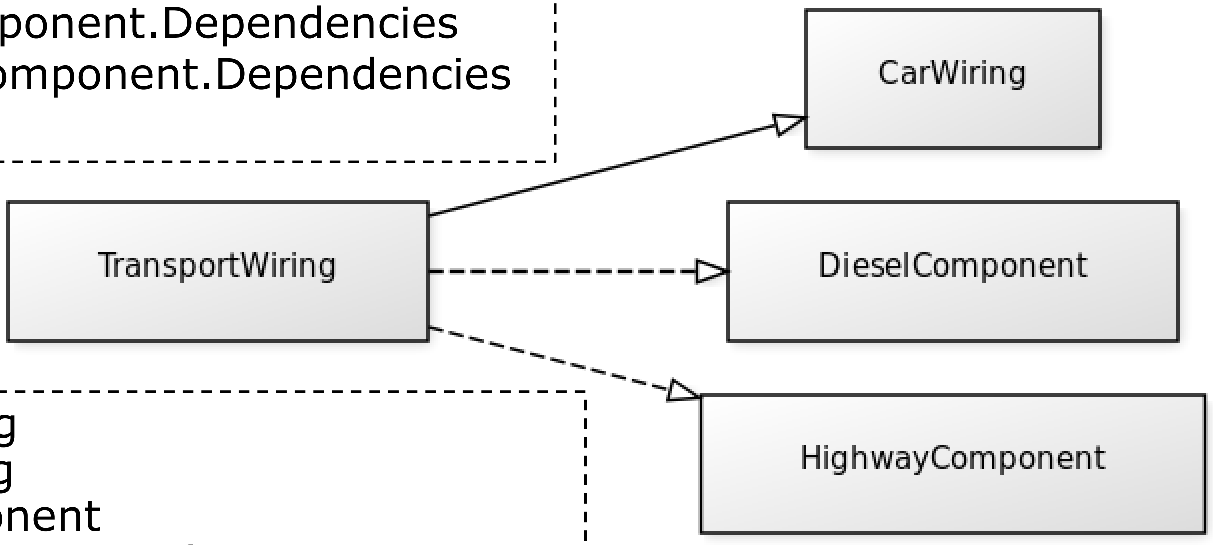
```
object CarWiring {
   type Dependencies = CarComponent.Dependencies
}
```

```
trait CarWiring extends CarComponent {
   self: CarWiring.Dependencies =>

   lazy val vehicle = new Car
}
```

There is no guarantee that the dependencies have been instantiated at this point. Therefore, use lazy val to avoid null pointer exception.

Multiple Component Wiring

```
object TransportWiring {
   type Dependencies =
      CarWiring.Dependencies
      with DieselComponent.Dependencies
      with HighwayComponent.Dependencies
}
```

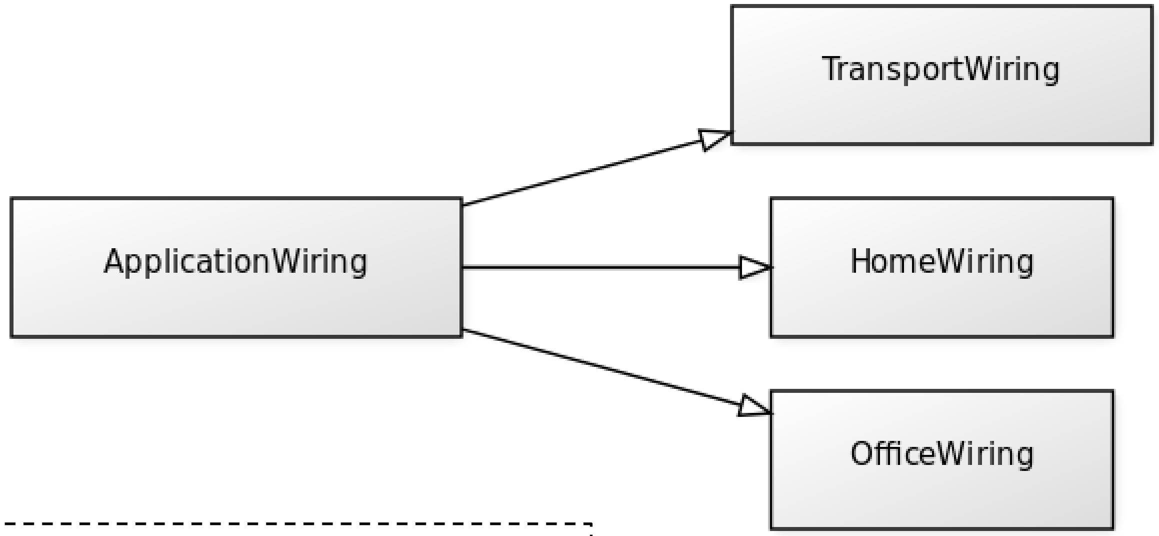


```
trait TransportWiring
   extends CarWiring
   with DieselComponent
   with HighwayComponent {
   self: ModuleWiring.Dependencies =>

   lazy val fuel = new Diesel
   lazy val road = new Highway
}
```

Wiring

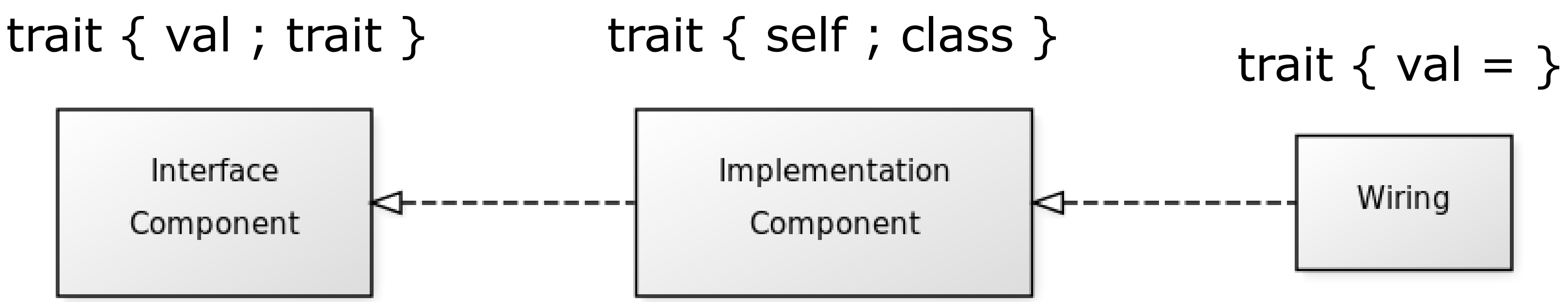
- Do not wire in a Component class.
- Do not implement in a Wiring class.
- Wiring is **programmatic configuration**.

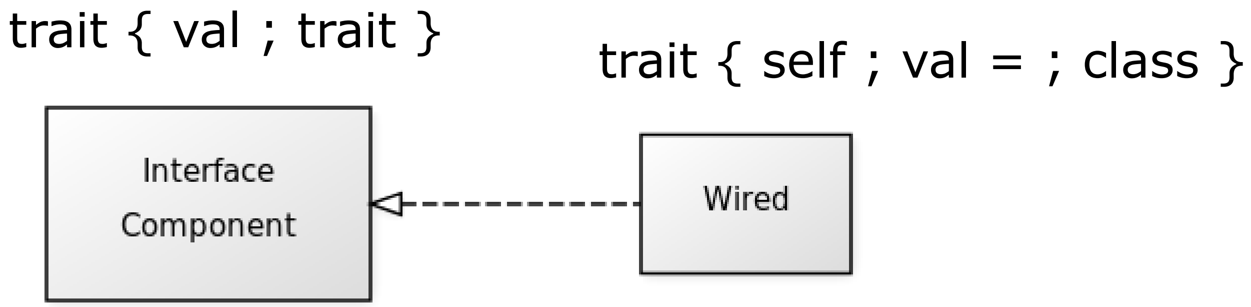
Application Wiring

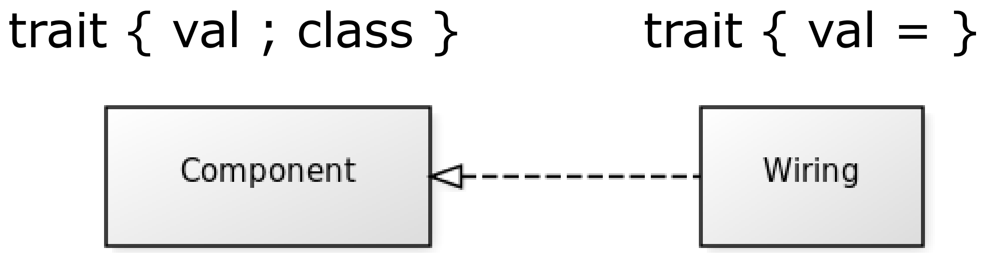


```
class ApplicationWiring
    extends TransportWiring
    with HomeWiring
    with OfficeWiring
```

Mixing the Cake







Wired = Implementation + Wiring

```
object CarWired {
   type Dependencies =
      FuelComponent with RoadComponent
}
```

Must extend the component interface

```
trait CarWired extends VehicleComponent {
   self: CarWired.Dependencies =>

   lazy val vehicle = new Car

   class Car extends Vehicle {
      fuel.##
      road.##
   }
}
```

Wiring

Implementation

Mock with Mockito

```
class TestWiring
    extends CarWiring
    with FuelComponent
    with RoadComponent {
    lazy val fuel = mock[Fuel]
    lazy val road = mock[Road]
}

new TestWiring {
    vehicle.##
    verify(fuel).##
    verify(road).##
}
```

Calling the hash of vehicle causes the car to be initialized which in turn calls the hash of fuel and road.

Scope

Singleton Scope

```
trait PlanetComponent {
    val planet: Planet
}
```

No Scope

```
trait FoodComponent {
    def food: Food
}
```

Managed Scope

```
trait WithConnectionComponent {
    def withConnection[T](block: Connection => T): T
}
```

Context Scope

```
trait ServiceComponent {
    def service(implicit context: Context)

    trait Service
}
```

Wiring

```
trait HealthServiceWired extends ServiceComponent {
    def service(implicit context: Context) = new HealthService

    class HealthService(implicit context: Context) extends Service
}
```

Implementation

Aspect-Oriented Programming (AOP)

```
trait TransactionalComponent {
    def transactional[T](block: => T): T
}
```

Call-by-Name effectively extends the grammar of Scala.

Cake Pattern usage:

```
def add = transactional {1 + 2}
```

Spring Annotation usage:

```
@Transactional def add = 1 + 2
```

Don’t eat too much cake!

Define simple injectables with no dependencies as outer classes rather than as inner classes of a component.

```
trait Clock {
    def read: Long
}

object SystemClock extends Clock {
    def read = System.currentTimeMillis
}
```

```
trait ClockComponent {
    implicit val clock: Clock
}

trait SystemClockWiring extends ClockComponent {
    val clock = SystemClock
}
```

Implicit Sub-Injection

```
case class Ticket(film: String, purchaseTime: Long)

object Ticket {
    def apply(film: String)(implicit clock: Clock) =
        new Ticket(film, clock.read)
}
```

```
trait CinemaComponent {
    self: ClockComponent =>

    val cinema: Cinema

    class Cinema {
        def buyTicket(film: String) = Ticket(film)
    }
}
```

Set Up And Tear Down Hooks

```
trait SetUpHookComponent {
    def setUpHook(hook: => Unit)
}
```

```
trait SetUpHookWired {
    private var setUpHooks = List.empty[() => Unit]

    def setUpHook(hook: => Unit) {
        setUpHooks ::= (() => hook)
    }

    def setUp() {
        setUpHooks.foreach(_())
    }
}
```

Actor

```
Object EchoActor {
    type Dependencies = ListenComponent
}
```

```
class EchoActor(injector: EchoActor.Dependencies) extends Actor {
    def receive = {
        case message: Message =>
            injector.listen.##
            sender ! message
    }
}
```

Call hash on listen and echo message

Props Wiring

```
trait EchoPropsComponent {
   val echoProps: Props
}
```

Notice that the type is only Props

```
object EchoPropsWiring {
   type Dependencies = EchoActor.Dependencies
}

trait EchoPropsWiring extends EchoPropsComponent {
   self: EchoPropsWiring.Dependencies =>

   val echoProps = Props(new EchoActor(self))
}
```

A simple single line function is just wiring!

ScalaTest, Mockito & Akka TestKit

```
class EchoActorTest extends WordSpec with Matchers with MockitoSugar {
  "An echo actor" should {
    "echo a message" in {
      new TestKit(ActorSystem("EchoActorTest"))
         with EchoPropsWiring with ListenComponent {
         val listen = mock[Listen]
         val message = new Message
          val actor = system.actorOf(echoProps)
         val probe = TestProbe()
         actor.tell(message, probe.ref)
         probe.expectMsg(message)
          verify(listen).##
          TestKit.shutdownActorSystem(system)
      }
    }
  }
}
```

ImplicitSender can be used to eliminate explicit TestProbe.

Actor Wiring

```
object EchoActorWiring {
   type Dependencies =
      EchoPropsComponent
      with ActorFactoryRefComponent
      with SetUpHookComponent
}
```

```
trait EchoActorWiring {
   self: EchoActorWiring.Dependencies =>

   lazy val echoActor = actorFactoryRef.actorOf(echoProps, “Echo")

   setUpHook {
      echoActor.##
   }
}
```

Use set up hook to ensure echo actor is started after application wiring is complete.

Conventions

- One access point per component.
- Component, Wiring, Wired suffices.
- Type aliases for dependencies.
- At least a 2 layer cake.

Why

- Easier to work effectively in a team.
- Easier to track down wiring problems.
- Easier to extend and rewire.

- Shared Game Services
- Peter Potts
- Principal Architect
- ppotts@kixeye.com
- Any questions?