Exercise 6: Family Polymorphism

TECHNISCHE UNIVERSITÄT DARMSTADT

Software Engineering Design & Construction
WS 2016/17 - Dr. Michael Eichberg, M.Sc. Matthias Eichholz

Although this exercise is not graded, it is highly recommended to also do them on your own. Just looking at a solution is much easier in comparison to actually coming up with it. Support can be found in the forum:
`https://www.fachschaft.informatik.tu-darmstadt.de/forum/viewforum.php?f=234`

Task 1 Virtual Classes

In this exercise, you will revisit the general, abstract framework for different kinds of card games, developed in the previous exercise (Exercise 5). The goal of this exercise is to reimplement the abstract framework using Virtual Classes. Since Scala lacks support for deep, nested mixin composition, we are going to use a small library providing support for Virtual Classes in Scala. This library allows us to define families of related classes, which can be mixed and matched on-demand.

A family of classes can be created by annotating a class with `@family`. Virtual classes can then be defined inside of a family using the `@virtual` annotation. The following example creates a family `FamilyA` and one virtual class `VirtualClassA`.

```
@family class FamilyA {
  @virtual abstract class VirtualClassA {
    def a: Int
  }
}
```

By extending a family, new virtual classes can be added or existing virtual classes can be extended. In the following example, a new method `aa` is added to the virtual class `VirtualClassA`.

```
@family class FamilyB extends FamilyA {
  @virtual override abstract class VirtualClassA {
    def aa: Int
  }
}

@family class FamilyC extends FamilyB {
  @virtual override class VirtualClassA(val a: Int) {
    def aa: Int = 2
  }
}
```

Implementing all abstract class members allows a virtual class to become a concrete class as shown in `FamilyC`. Using the previously defined virtual classes, the following program prints the numbers 5 and 2.

```
object Demo extends App {
  val family = FamilyC()
  val virtual = family.VirtualClassA(5)
  println(virtual.a)
  println(virtual.aa)
}
```

Similar to Scala's Traits, new families can be created by mixing multiple families together. The program `Demo2` produces the numbers 3 and 4 as output.

```
@family class FamilyD extends FamilyB {
  @virtual override abstract class VirtualClassA {
    def aa: Int = 4
  }
}
```

```
@family class FamilyE extends FamilyA {
  @virtual override class VirtualClassA {
    def a: Int = 3
  }
}

@family class FamilyF extends FamilyD with FamilyE

object Demo2 extends App {
  val family2 = FamilyF()
  val virtual2 = family2.VirtualClassA()
  println(virtual2.a)
  println(virtual2.aa)
}
```

Implement the composable class hierarchy that models different aspects of playing cards, starting from the following base family:

```
@family class CardsModel {

  @virtual abstract class Cards {
    def deck: Seq[Card]
    val cardOrdering: Ordering[Card]
  }

  @virtual abstract class Card {
    def name: String

    override def toString: String = name
  }
}
```

Do not modify the base family in your implementation. Use virtual classes to model `SuitedCards` and `RankedCards` and compose them into `SuitedAndRankedCards`. As before, `FullRankedCards` and `PiquetRankedCards` should fix the ranks to the standard 52 (2 to Ace) and 32 (7 to Ace) ranks, respectively and `FrenchSuitedCards` should fix the suits to the four suits available in a French deck. Finally, create a `FullFrenchCards` family and use it to implement a MauMau game similar to the previous exercise.

The template we provide for this exercise consists of three SBT sub-projects `cards`, `examples` and `macros`. Add your implementation to the `cards` sub-project. You can compile and run your implementation using `sbt compile` and `sbt run` respectively. Within the `examples` sub-project you will find various examples on how to use the Virtual Traits library, amongst others the smart home example presented in the lecture. You can run the examples using `sbt examples/run`. Don't expect to get any support from your IDE with respect to type checking or code completion. It is recommended to write the code using a text editor and compile/run it using SBT.

Further details explaining the concept of Virtual Classes in Scala in more detail can be found in the corresponding paper[1].

Task 2 SOLID

Assess your design with respect to the five SOLID principles. Think about whether the principles also apply for a family of classes and not only for a single-class hierarchy.

Task 3 Documentation

Suppose we want to build a collection class inheriting from `scala.collection.Traversable` that is able to report the number of executed operations for each method. Which method calls can cause problems with the reported number of executed operations when instrumenting all available methods with the reporting mechanism? What are the reasons for these problems? Hint: Also have a look at the super classes like `scala.collection.TraverableOnce`.

[1] `http://dl.acm.org/citation.cfm?id=2637654&CFID=872172699&CFTOKEN=39009737`