Exercise 7: Advanced Inheritance and Analyzing Code for Design Patterns

TECHNISCHE UNIVERSITÄT DARMSTADT

In the first part of the exercise, you will design and implement parts of a general, abstract framework for different kinds of card games and then instantiate it for specific examples. Your solution should be as simple as possible. As a first step, make yourself familiar with the code. Add your implementations in the corresponding places and test them thoroughly. As a minimum requirement, your code must at least compile without errors.

In the second part of this exercise, you will analyze an existing code base for design patterns. Submit your solution as a PDF.

Task 1 The Cards Layer

See sbt project.

Task 2 The Base Layer

See sbt project.

Task 3 Safety

Explain in a comment of your `Game` trait why a player cannot play cards from a wrong card deck. For example, in a game of Mau Mau with a Standard 52 deck, a player cannot play cards from another French, German or Uno deck. What exactly prevents a player from doing so?

Task 4 In Java?

Think about how you would design your framework in Java 7. Add a comment to your Scala `Game` trait that names the difficulties that would arise compared to the Scala version. Hint: think about the features of Scala you use in your solution and how your design benefits from them.

Task 5 Second Part: Scala 2.7 Collections

In the project for the second exercise, you will find the source code for the Scala 2.7 standard library, which contains a collection hierarchy. Note: do not expect to be able to compile it with a current Scala compiler or in Eclipse. We know that Scala 2.7 is an old version, but newer versions are too complex, please do not try to analyze other Scala versions than 2.7. Analyze how the core of the collections hierarchy makes use of the following design patterns:

- Factory (any variant: abstract factory, factory method)
- Observer
- Strategy
- Template

Task 5.1 Factory Method (1P)

Instances of this pattern can be found in many companion objects, most commonly the `apply` methods and `empty` methods, e.g.,

```
def apply[A](xs: A*): Seq[A]
```

in `scala.Seq`, or

```
def empty[A, B]: Map[A, B]
```

in `scala.collection.Map`[1], but also

```
def fromIterator[T](source: Iterator[T]): PagedSeq[T]
def fromIterable[T](source: Iterable[T]): PagedSeq[T]
def fromStrings(source: Iterator[String]): PagedSeq[Char]
def fromReader(source: Reader): PagedSeq[Char]
...
```

and alike in `scala.collection.PagedSeq`.
The above mentioned methods are static factories, which create concrete collections as products which adhere to a base collection trait. For example, `Map.apply` from package `scala.collection.mutable` creates a `HashMap` instance, which extends the abstract `scala.collection.mutable.Map` trait.

Task 5.2 Observer

Instances of this pattern can be exclusively found in the observable collections sub-hierarchy in package `scala.collection.mutable`, e.g., `ObservableSeq`. The subjects are defined by base trait `Publisher`, concrete subjects are the observable collections, the observers are defined by base trait `Subscriber`, concrete subscribers are passed to the subscription methods defined in trait `Publisher`. Observable collections notify registered subscribers when and which elements are added and removed.

Task 5.3 Strategy

Strategies are very common as well. There are two categories of strategies:

- Function arguments of `map`, `filter`, `foldLeft` and many similar methods in all common collections. The abstract strategy interface is defined by Scala functions. The concrete strategy implementations are usually instantiated at call site as anonymous closures.
- `Ordered` instances for `min`, `max` and sorting methods. The abstract strategy interface is defined by `Ordered` trait. The concrete strategy implementations are usually defined a priori and many of them can be found in the standard library, e.g., in `scala.Predef`.

For both categories, the contexts are the method calls (not objects instantiated with strategies as in the standard text book strategy pattern) of map, filter, min, max etc. The method implementation determine that basic structure of an algorithm, the strategies are used to parameterise that algorithm with details, e.g., to which objects elements are mapped or how they are compared.

Task 5.4 Template

There are many template methods in the collections hierarchy. Two in the topmost trait are `Iterable.sameElements` and `Iterable.mkString`. Many other traits and classes use template methods.
For template method `Iterable.sameElements`, the abstract class (trait) is `Iterable`, a concrete class is for example `scala.List`. Trait `Iterable` uses *primitive* method `elements` to implement many template methods such as `sameElements`. A concrete subclass has to define how to iterate through its elements and gets many (template) methods "for free".

Task 6 Fragile Base Classes

Can you find an instance of the Fragile Base Class problem in the collections hierarchy? More precisely, can you find either an instance of the problem,

- where everything works as expected, but where the behavior might break in the future because of a fragile base class, or
- one where a method already behaves in a probably unexpected manner because it expects a self-call structure different from the one that is in place (potentially because the base class behavior *has been* changed previously).

Say which classes are affected and explain which problems might or already do occur. Only consider classes and traits that are already present in the hierarchy. You are not allowed to invent new classes, except for the purpose of making traits concrete. You are not allowed to create new classes that override existing methods (otherwise you could easily introduce the problem yourself).
Note: in contrast to the previous task, this analysis does not have to be complete. A single example of either one of the above instances is enough.

[1] Note that covariant immutable collections define an `Empty` *value* instead of an `empty` method, which are *singletons* and not factories.

Task 6.1 Solution

One instance of the FBCP is already present in `ObservableSet` extending `Set`. Method `ObservableSet.clear()` calls `super.clear` and publishes a reset message. In a concrete instance, the super call can get dispatched to `Set.clear`, which calls `-=` for each element. However, `Set.-=` is overridden/implemented in `ObservableSet` and publishes a remove message for each element. In effect, `ObservableSet.clear` publishes both a reset and a clear for every element, which probably not expected.