Exercise 7:
Design Patterns

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 office hours of our tutors and in the forum at `https://www.fachschaft.informatik.tu-darmstadt.de/forum/viewforum.php?f=234`.

Task 1 Scala 2.7 Collections

In the project for the first 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 / IntelliJ. 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

Look at all of the following classes and traits and their corresponding companion objects:

- Everything in packages `scala`, `scala.collection` and `scala.collection.mutable` that directly or transitively inherits from `scala.Iterable`.
- All base traits and base classes of the above traits and classes *only if they participate in a pattern*.

For each pattern name the participants, which roles they have, how they collaborate and how they might vary from the standard text book examples. Explain the pattern in terms specific to the current pattern instance, e.g., don't just copy text from a design patterns book. Your analysis should be complete and you should mention how often a specific pattern is used in comparison to others. If a pattern is used in a similar way in many different places, give a detailed analysis of a single example and say how other instances of the pattern in the collection hierarchy relate to it.

Task 1.1 Factory Method

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]
...
```

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

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 1.2 Observer

Instances of this pattern can be exclusively found in the observable collections sub-hierarchy in the 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 1.3 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 1.4 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 2 Synchronized Maps

Now have a look at the `SynchronizedMap` trait in the package `scala.collection.mutable`. Which design pattern is used to add synchronization to maps? Does the design pattern that is used resolve the fragile base class problem? Justify your answer. For this task you can use the current version of the Scala standard library.

Task 2.1 Decorator Pattern

The static variant of the decorator pattern is used to add synchronization to maps. Using the decorator pattern does not resolve the fragile base class problem completely. Examples can be found in the lecture slides on the Decorator Pattern.