LOG6306
TP1 Design Patterns

Rodrigo Morales DGIGL
Polytechnique Montréal, QC, Canada
rodrigomorales2@acm.org

Due date before 16th of November 2018 at 11:59 pm

1 Rule

All exercises must be done in teams.

2 Goal

Study and understand the concrete implementation of the design patterns studied in class: Abstract Factory, Composite, Iterator, Observer, Singleton, and Visitor (excluding Decorator).

3 Evaluation criteria (full mark: 60)

- 5 points if the design pattern instance is a true instance of the design pattern declared by the team (for each pattern)
- 5 points if the description, UML diagram is correct (for each pattern)

3.1 Format

- You need to submit a report in pdf format including a cover page with the name of each team's member, matricule and the information required in the following paragraphs.
- Submit your report to the email above

3.2 To do:

1. For each of the six design pattern, list their roles and recall their definitions.
2. By exploring the open-source projects in the SF110 dataset (http://www.evosuite.org/experimental-data/sf110/), find and explain one example of each of the six design patterns mentioned above using an extended UML class diagram. You also need to provide the code of classes participating in the design pattern in the report. Take as an example the one provided for the abstract factory pattern from the book Patterns of Java from Mark Grand.

factory class it uses is determined by initialization code, and the singleton concrete factory object is returned by its `getDefaultToolkit` method.

Code Example

For this pattern's code example, we will return to the problem discussed under the Context section. Figure 5.8 shows an expanded class diagram that incorporates the Abstract Factory pattern.

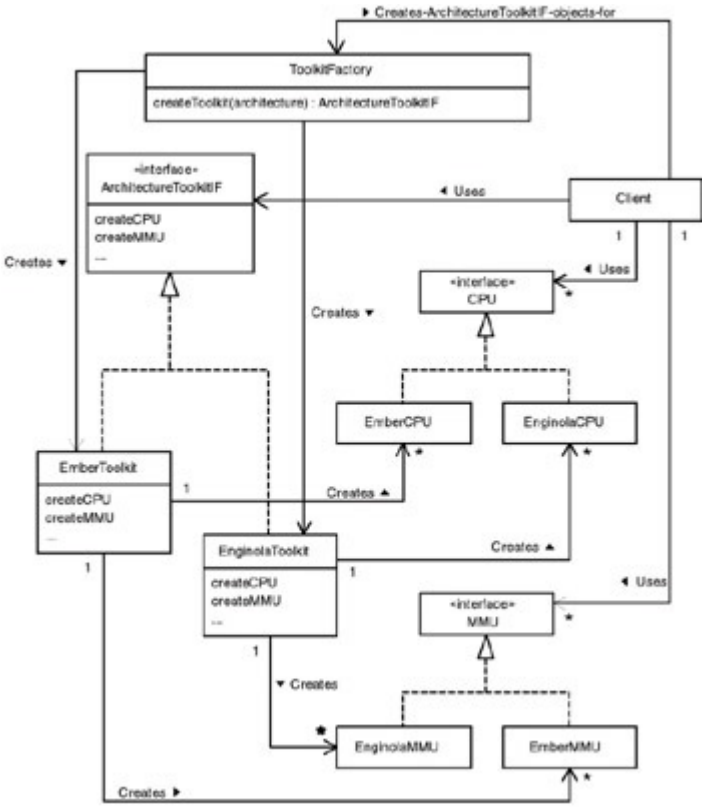

Figure 5.8: Diagnostic classes with abstract factory.

An instance of the `Client` class manages the remote diagnostic process. When it determines the architecture of the machine it has to diagnose, it passes the architecture type to the `createToolkit` method of a `ToolkitFactory` object. The method returns an instance of a class such as `EmberToolkit` or `EnginolaToolkit` that implements the `Architecture-ToolkitIF` interface for the specified computer architecture. The `Client` object can then use the `ArchitectureToolkitIF` object to create objects that model central processing units (CPUs), memory management units (MMUs), and other components of the required architecture.

Here is some of the Java code that implements the design for remote computer diagnostics shown in Figure 5.8. The abstract widget classes have the obvious structure.

This code is for a concrete factory class that creates instances of classes to test ember architecture computers:

```
class EmberToolkit implements ArchitectureToolkitIF {
    public CPU createCPU() {
        return new EmberCPU();
    }  // createCPU()

    public MMU createMMU() {
        return new EmberMMU();
    } // createMMU()
    ...
} // class EmberFactory
```

The following is the code for the factory interface:

```
public interface ArchitectureToolkitIF {
    public abstract CPU createCPU() ;
    public abstract MMU createMMU() ;
...
}  // AbstractFactory
```

This implementation of the Abstract Factory pattern uses the Factory Method pattern to create `ArchitectureToolkitIF` objects. Here is a listing of the factory class responsible for creating `ArchitectureToolkitIF` objects.

```
public class ToolkitFactory {
    /**
     * The single instance of this class.
     */
    private static ToolkitFactory myInstance
      = new ToolkitFactory();

    // Symbolic names to identify computer architectures
    public final static int ENGINOLA = 900;
    public final static int EMBER    = 901;
    // ...

    public static ToolkitFactory getInstance() {
        return myInstance;
    } // getInstance()

    /**
     * Return a newly created object that implements the
     * ArchitectureToolkitIF interface for the given computer
     * architecture.
     */
```

```
    public
    ArchitectureToolkitIF createToolkit(int architecture) {
        switch (architecture) {
          case ENGINOLA:
              return new EnginolaToolkit();

          case EMBER:
              return new EmberToolkit();
        }  // switch
        String errMsg = Integer.toString(architecture);
        throw new IllegalArgumentException(errMsg);
    }  // createToolkit(int)
}  // class ToolkitFactory
```

Client classes typically create concrete widget objects using code that looks something like this:

```
public class Client {
    public void doIt () {
        ToolkitFactory myFactory;
        myFactory = ToolkitFactory.getInstance();
        ArchitectureToolkitIF af;
        af = myFactory.createToolkit(ToolkitFactory.EMBER);
        CPU cpu = af.createCPU();
        ...
    }  // doIt
}  // class Client
```

Related Patterns

Factory Method. In the preceding example, the abstract factory class uses the Factory Method pattern to decide which concrete factory object to give to a client class.

Singleton. Concrete Factory classes are usually implemented as Singleton classes.