Introduction to Software Engineering

Assignment 7

Dimitri Tabatadze, 2024-12-10

1 Design Patterns in Java API

Specify the design pattern of the following Java API classes/ interfaces:

a) Class `java.io.Reader` with subclasses `BufferedReader`, `FilterReader`, `PushbackReader`.
b) Method `getInstance()` of class `java.util.Calendar`.
c) Method `valueOf(int)` of class `java.lang.Integer`.

For each of the found design patterns, draw a UML class diagram that reflects the section of the Java API with the design pattern. In the UML class diagrams, limit yourself to the methods and attributes relevant to the design pattern. Specify which of the classes, methods, and attributes in your UML class diagram correspond to which roles in the design pattern. Use the terms from the lecture. Explain in one sentence how you recognized this pattern.

Solution

a) The design pattern used in the given instance is *Decorator*.

   “For example,

   ```
   BufferedReader in
     = new BufferedReader(new FileReader("foo.in"));
   ```

   will buffer the input from the specified file.” — java documentation

   “Abstract class for reading filtered character streams. The abstract class FilterReader itself provides default methods that pass all requests to the contained stream” — java documentation

   “A character-stream reader that allows characters to be pushed back into the stream.” — java documentation

   In all cases, the subclass implements additional features, while taking another reader as an input and acting on the given reader.

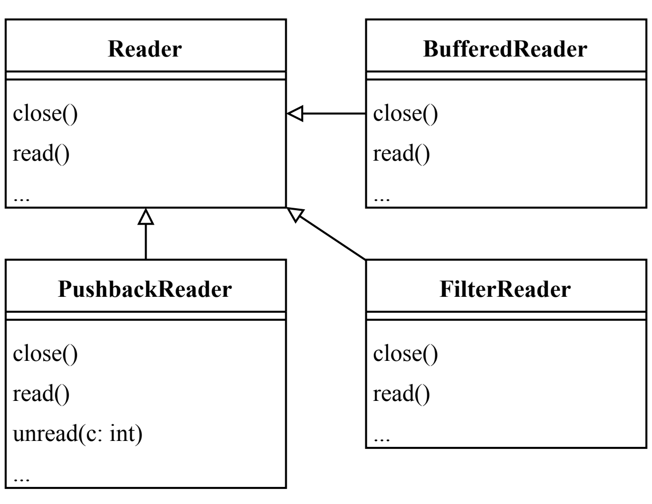

Figure 1: the thing

b) The design pattern used in the given instance is *Factory*.

   “Like other locale-sensitive classes, Calendar provides a class method, getInstance, for getting a generally useful object of this type. Calendar's getInstance method returns a Calendar object whose calendar fields have been initialized with the current date and time:

   ```
   Calendar rightNow = Calendar.getInstance();
   ```

   ” — java documentation

   The `getInstance()` method creates a new Calendar subclass object and returns it. The subclass used depends on the locale defaults.

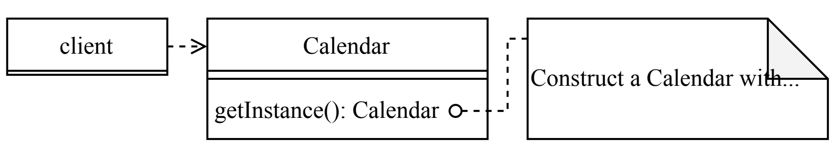

Figure 2: basic structure of `java.lang.Calendar`

c) The design pattern used in the given instance is *FlyWeight*. The java documentation says the following about the implementation of `Integer.valueOf(int)`

   “Returns an Integer instance representing the specified int value. If a new Integer instance is not required, this method should generally be used in preference to the constructor Integer(int), as this method is likely to yield significantly better space and time performance by caching frequently requested values. This method will always cache values in the range −128 to 127, inclusive, and may cache other values outside of this range.” — java documentation.

   This clearly matches the description of FlyWeight.

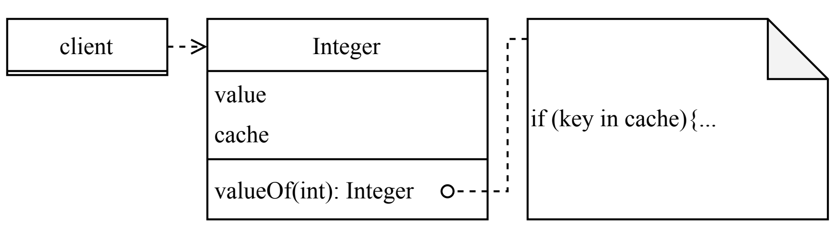

Figure 3: basic structure of `java.lang.Integer`

2 Queueing Simulation

Provide a design for the waiting area of a queueing simulation. In a queueing simulation, tasks are entered into the waiting area (the “queue”) and removed one by one by a server that processes them. There are three ways how tasks are handled in the waiting area:

a) A queueing discipline (first come, first served)
b) A stack discipline (last come, first served)
c) A priority queue: the first task with the highest priority is served first.

The queueing simulation must be able to choose among these three possibilities dynamically. Provide a UML diagram for the waiting area. Mark in the diagram, which design pattern(s) you are using. Also, the design must include the classes `java.util.Queue<E>` and `java.util.Stack<E>`. These do not have to be re-implemented, but they need to be properly integrated (using a pattern).

Solution

We calearly need to implement a decorator design pattern.

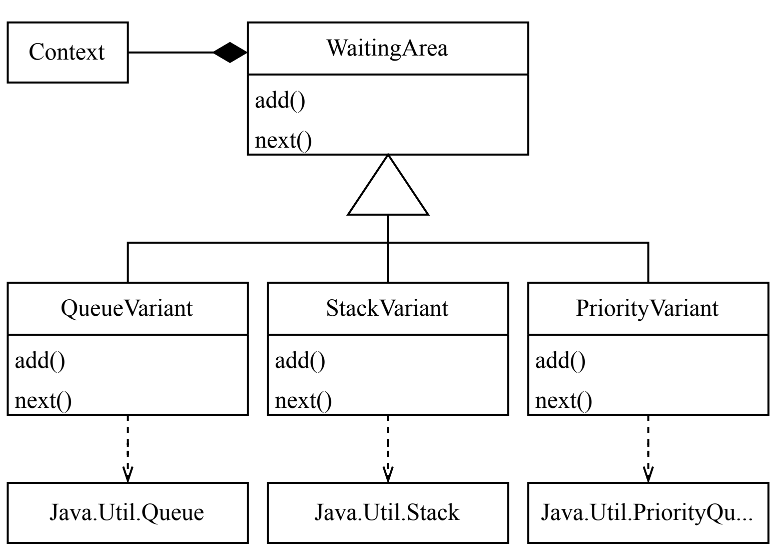

Figure 4: my implementation

3 Coffee shop

Imagine you own a small coffee shop. Like most beginners, you start off with a simple menu: just two types of coffee — house blend and dark roast. In your billing system, you have one class per coffees, inheriting from an abstract “beverage” class. Things go well as customers trickle in, enjoying your (admittedly bold) coffee. But soon, some customers — perhaps new to coffee — begin asking for milk or sugar. An insult to pure coffee, perhaps, but customer satisfaction is paramount! Now you need to add these options to the menu and, unfortunately, to your billing system. At first, your IT person creates a subclasses for milk and sugar for each coffee class. But then, as expected, a customer comes along with a dreaded request: “Could I get a milk coffee with sugar, please?” The billing system, not prepared for this, throws a fit. So, it's back to the drawing board.

The IT team modifies the code again, creating a subclass to handle milk coffee with sugar for each type of coffee. Things settle down, business picks up, and customers seem happy. It seems like smooth sailing — until your new competitor opens across the street. This new shop offers more than 10 coffee options, with a whole lineup of add-ons, such as foamy milk and artificial sweetener! Wanting to keep up, you expand your offerings but realize there's no way your current system can handle every possible combination of coffees and add-ons without becoming overly complex.

a) Identify which design pattern will help to solve the problem without making the billing system overly complex.
b) Draw class diagram according to the scenario including the design pattern from Q1.
c) Include elegant code to compute the cost of the coffee with all the chosen options.

Solution

Since we have to implement extensions as we go, we will have good time using a Decorator pattern.

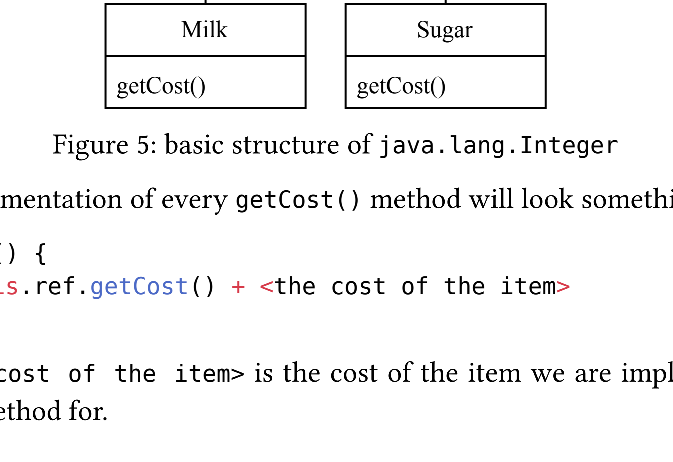

Figure 5: basic structure of `java.lang.Integer`

and the implementation of every `getCost()` method will look something like

```
int getCost() {
  return this.ref.getCost() + <the cost of the item>
}
```

where `<the cost of the item>` is the cost of the item we are implementing the `getCost()` method for.