Java I/O Report

Elais Jackson

October 17, 2013

1 Introduction

For our example of Java I/O in the context of software engineering principles, we will focus on two aspects of the library. First we will discuss how the use of decorators in Java I/O adhere to good software engineering principles in general. Then we will become more critical and talk about a case where java.io.Reader violates the Liskov Substitution Principle(LSP). These two things should make for a nice introduction to looking at Java.IO under the guise of software engineering principles.

The principles that lie at the heart of the object-oriented paradigm are the SOLID principles of: class design. SOLID is shorthand for Single responsibility principle(SRP), open/closed principle(OCP), Liskov substitution principle(LSP), Interface segregation principle (ISP), and dependency-inversion principle(DIP. Taken together they allow for one to build remarkably robust software systems that take into account other coders having to read the code and reduces the overall complexity. It is within this context that we look at java I/O and all of our report will focus on aspects of the package that demonstrate one of the aforementioned software engineering principles.

2 Java.IO Examples

2.1 The Decorator Pattern

Decorators allow class behavior to be extended dynamically at runtime and are used extensively in Java I/O stream. Decorators are used to form large object structures across many disparate objects. Usually, subclassing is considered to be the best way to approach this but in some cases subclassing might be impractical. What makes decorators more desirable is that they adhere to the open/closed principle where classes should be open for extension, but closed for modification. Usually there is some component that defines the interface for objects and can have responsibilities added dynamically, and another component that can be made that is an implementation of this interface. Decorators essentially wrap these new components, adding functionality to existing objects like classes defined in a library.

Take for example the *InputStream* abstract class in java i/o. Concrete implementations such as *FileInputStream*, *BufferedInputStream*, and *ObjectInputStream* all include a constructer that takes an instance of it. All methods delegate to the wrapped instance with changes in the way they behave to tackle more specific problems that they may be dealing with. So lets say we have data in a .dat file and that we want to read them quickly, the code would look something like this:

Listing 1: Some Code

```
public class JavaIO {
    public static void main(String[] args) {
        // Open an InputStream.
        FileInputStream in = new FileInputStream("test.dat");
        // Create a buffered InputStream.
        BufferedInputStream bin = new BufferedInputStream(in);
        // Create a buffered, data InputStream.
        DataInputStream dbin = new DataInputStream(bin);
        // Create an unbuffered, data InputStream.
        DataInputStream din = new DataInputStream(in);
        // Create a buffered, pushback, data InputStream.
        PushbackInputStream pbdbin = new PushbackInputStream(dbin);
    }
}
```

What we just did here was open an *inputstream* of the file, buffered it into memory, then created an unbuffered version, and used many implementations of *inputstream* without having to change the data we use or have particuclar knowledge of each module other than how they extend *inputstream*. A case could also be made that LSP is being followed but we discuss that principle more in depth later. The benefit from using decorators over subclassing in this case was that one does not have to have a class for every possible combination of the file input they may have gotten. Another added benefit of using decorators is you only have to close the outermost decorator to close the stream. This adds to our action economy and saves us extra steps.

2.2 Reader

We also have the class java.io.Reader. *Reader* is a base class for character-oriented input streams. There are two methods, *mark* and *reset*, that are used to mark the current position in a stream and to return to that position later. However, a key problem arises from the fact that derived classes do not have to support *mark* and *reset*. Lets say one programmer were to write a function that calls *Reader*, lets call it *newReader*, then another programmer writes a function that calls *newReader*. If the programmer who wrote a function that called *newReader* didnt know that it required a *Reader* that supported *mark* and *reset*, why would he check the return value for success?

This violates the Liskov substitution principle. Because derived classes don't have to support all of the methods available in Reader, the rule that a programmer should be able to use a given class and its derived classes interchangeably with no loss in power is broken. If a designer is to make a base class with optional methods, a good question to ask is: why not make two base classes where all methods are in each by default?

3 Conclusion

In conclusion we have given two examples of software engineering principles either being implemented correctly or violated by examples from Java IO. In the first example we showed how the open/closed principle was preserved by the creative use of decorators to extend functionality without having to write new subclasses. In essence, it demonstrated perfectly the idea of being open for extension but closed for modification. In the second example we talked about two methods in the Reader class that don't necessarily have to be supported by derived classes. This violated the Liskov substitution principle by having changes in the base class that affects the client without the client's knowledge, unless the user had prior knowledge of that module. So the client would have to have specific knowledge of the derived methods he's using from the base class to insure that he does not throw an error. In a system that was true to LSP, he would not have to have any knowledge of the base class's differences from its derived classes to implement them correctly. These two examples show perfectly both use and misuse of software engineering principles in the java i/o class.