

# Design Principles in Test First Programming



objectmentor.com

by Erik Meade

[emeade@objectmentor.com](mailto:emeade@objectmentor.com)

## Introduction

The purpose of this article is to examine how test first programming produces code which adheres to certain design principles. In particular we will see that test first programming insures that we follow the [Open/Closed](#), [Liskov Substitution Principle](#) and [Dependency Inversion](#) principles, introduced to us in 1996 by Robert C. Martin. Here we will revisit the Copy program, which Robert used to demonstrate the Dependency Inversion Principle, only we will do so test first using Kent Beck and Erich Gamma's [JUnit](#), a testing framework for Java.

## The Copy program

Robert used the Copy program as an example of code rot. The basic Copy program reads from the keyboard and prints to the printer. Enhancing the Copy program so it can also print to the disk results in an if else statement based on type which definitely has a code smell to those of us who understand the Open/Closed Principle. As a reminder the Open/Closed Principle states that we should add new functionality by adding new code, not by editing old code.

Since the entire purpose of the Copy program is to copy, We will focus on writing the test for it first, but how do we write a test for a class that uses two other classes without creating those classes first? Do the simplest thing that could possibly work, create mock objects for the Reader and Writer classes. Now we have enough to implement test first. The MockReader will allow us to hard code what we are going to read, and the MockWriter will expose a field so we can check that it "wrote" the right thing.

```
import junit.framework.TestCase;

public class CopyTest extends TestCase {

    public CopyTest ( String name ) {
        super ( name );
    }

    public void testCopy () {
        String testLine = "input";
        MockReader reader = new MockReader ( testLine );
        MockWriter writer = new MockWriter ();
        Copy copy = new Copy ( reader, writer );
        copy.copy ();
        assertEquals ( testLine , writer.output );
    }
}
```

Compiling this produces a few errors, all related to the fact that none of the classes used exist. Addressing each error in the order they appear results in the following:

```
class MockReader {

    String input;

    MockReader ( String input ) {
        this.input = input;
    }

    public String readln () {
        String line = input;
        input = null;
        return line;
    }
}

class MockWriter {

    String output;

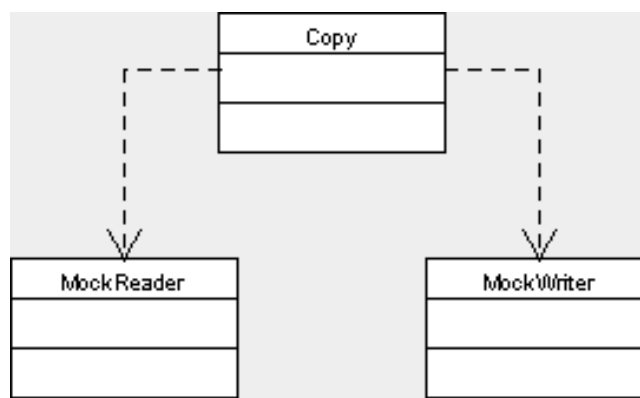
    public void writeln ( String line ) {
        output = line;
    }
}

class Copy {

    MockReader reader;
    MockWriter writer;

    Copy ( MockReader reader , MockWriter writer ) {
        this.reader = reader;
        this.writer = writer;
    }

    void copy () {
        String line;
        while ( ( line = reader.readln () ) != null )
            writer.writeln ( line );
    }
}
```



At this point our test for copy passes. We are not done yet however, we still have to implement the Reader and Writer classes. We could just change the Mock members and parameters in the Copy class, but that would render our test useless. We want to be able to substitute our mock objects for their counter-parts at any time. An interface will allow this to be archived.

```

interface Reader {
}

interface Writer {
}
  
```

Using interfaces here follows the Dependency Inversion Principle, which states:

- A. High level modules should not depend upon low level modules. Both should depend upon abstractions.
- B. Abstractions should not depend upon details. Details should depend upon abstractions.

Now we are going to break Copy, intentionally. We are going to use the compiler to tell us what we need to do. Searching and replacing "Mock" with nothing yields:

```

class Copy {

    Reader reader;
    Writer writer;

    Copy ( Reader reader , Writer writer ) {
        this.reader = reader;
        this.writer = writer;
    }

    void copy () {
        String line;
        while ( ( line = reader.readLine () ) != null )
            writer.println ( line );
    }
}
  
```

Compiling Copy gives us two "method not found in interface" errors. As we expect, since the Liskov Substitution Principle tells us: *Derived classes must be usable through their base class interface, without the client being able to tell the difference* and we have not defined the methods called on Reader and Writer in Copy. Copy and paste the signature

of these methods from the mock objects into their interfaces and change the brace to a semi-colon. The public key word can be deleted if you like, the default scope for an interface is public, but since that would be extra work I'll leave it.

Compiling and fixing such trivial errors may seem like extra work, but with larger programs "breaking" the program in small pieces keeps the number of errors down, and in cases similar to this one, insures that we only define the methods in our interfaces which are used. When clients do not depend on services they do not use we are following the Interface Segregation Principle.

```
interface Reader {
    public String readln ();
}

interface Writer {
    public void writeln ( String line );
}
```

Recompiling all the sources now reveals two "Incompatible type for constructor." errors in CopyTest. One for MockReader and one for MockWriter. This is what we would expect since the mock objects don't yet implement Reader and Writer. Implementing their perspective interfaces fixes these errors.

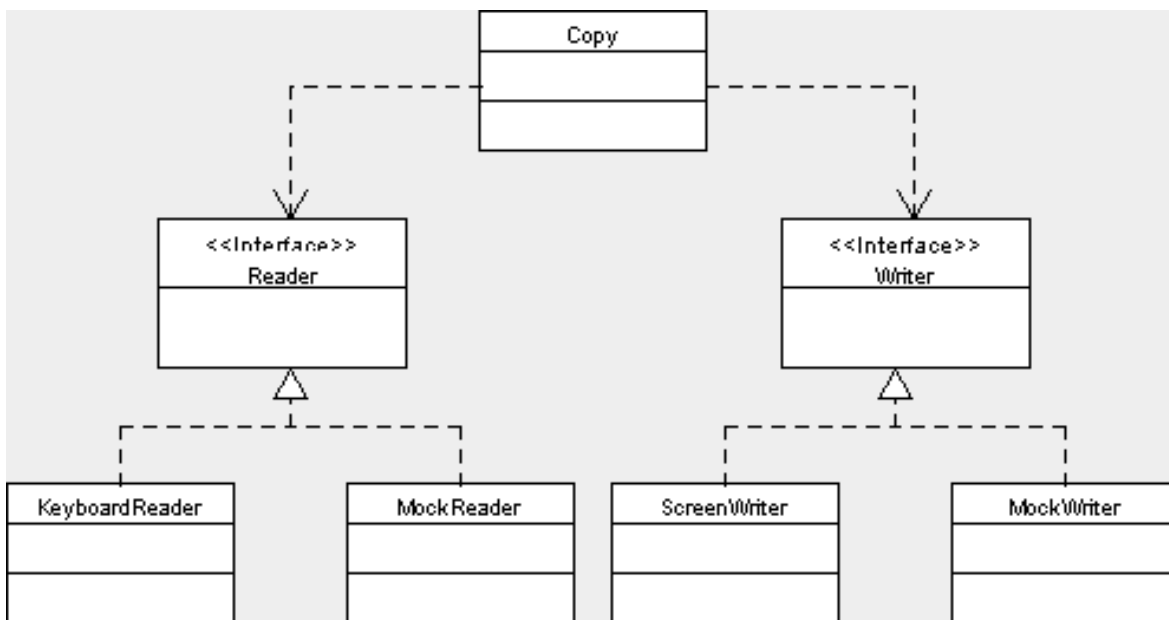
```
class MockReader implements Reader {
    String input;

    MockReader ( String input ) {
        this.input = input;
    }

    public String readln () {
        String line = input;
        input = null;
        return line;
    }
}

class MockWriter implements Writer {
    String output;

    public void writeln ( String line ) {
        output = line;
    }
}
```



Now, as the UML diagram suggests, we can implement our KeyboardReader and ScreenWriter tests, the result of which will be KeyboardReader and ScreenWriter each realizing their perspective interfaces.

## Conclusion

Examining the results of our test first programming version of Copy reveals that we are indeed following the Open/Closed, Liskov Substitution Principle and Dependency Inversion principles. Because our KeyboardReader and ScreenWriter are new realizations of the Reader and Writer interfaces, we are writing new code to add new functionality, an indicator that we are following the Open/Closed Principle. We are also following the Liskov Substitution Principle illustrated by the fact that the Copy program uses the Reader and Writer realizations through their base class interfaces. By comparing the first UML diagram with the second, we can see that test first caused the dependencies to be inverted, because testing through those dependencies requires the use of mock objects.