9

SRP: The Single Responsibility Principle

None but Buddha himself must take the responsibility of giving out occult secrets...

— E. Cobham Brewer 1810–1897.
Dictionary of Phrase and Fable. 1898.

This principle was described in the work of Tom DeMarco[1] and Meilir Page-Jones[2]. They called it *cohesion.* As we'll see in Chapter 21, we have a more specific definition of cohesion at the package level. However, at the class level the definition is similar.

SRP: The Single Responsibility Principle

THERE SHOULD NEVER BE MORE THAN ONE REASON FOR A CLASS TO CHANGE.

Consider the bowling game from Chapter 6. For most of its development the `Game` class was handling two separate responsibilities. It was keeping track of the current frame, and it was calculating the score. In the end, RCM and RSK separated these two responsibilities into two classes. The `Game` kept the responsibility to keep track of frames, and the `Scorer` got the responsibility to calculate the score. (see page 85.)

1. [DeMarco79], p310
2. [PageJones88], Chapter 6, p82.

Why was it important to separate these two responsibilities into separate classes? Because each responsibility is an axis of change. When the requirements change, that change will be manifest through a change in responsibility amongst the classes. If a class assumes more than one responsibility, then there will be more than one reason for it to change.

If a class has more then one responsibility, then the responsibilities become coupled. Changes to one responsibility may impair or inhibit the class' ability to meet the others. This kind of coupling leads to fragile designs that break in unexpected ways when changed.

For example, consider the design in Figure 9-1. The `Rectangle` class has two methods shown. One draws the rectangle on the screen, the other computes the area of the rectangle.

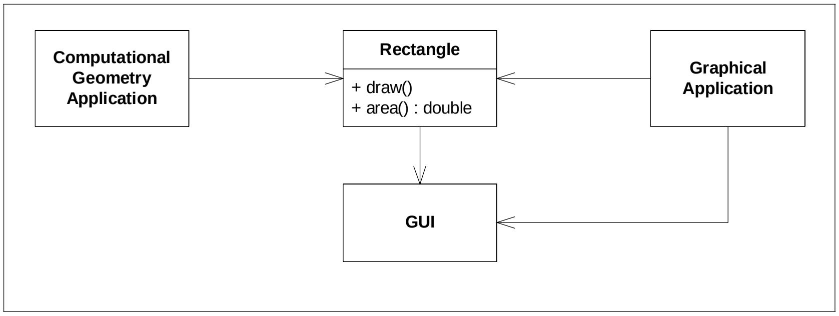

Figure 9-1
More than one responsibility

Two different applications use the `Rectangle` class. One application does computational geometry. It uses `Rectangle` to help it with the mathematics of geometric shapes. It never draws the rectangle on the screen. The other application is graphical in nature. It may also do some computational geometry, but it definitely draws the rectangle on the screen.

This design violates the SRP. The Rectangle class has two responsibilities. The first responsibility is to provide a mathematical model of the geometry of a rectangle. The second responsibility is to render the rectangle on a graphical user interface.

The violation of SRP causes several nasty problems. Firstly, we must include the `GUI` in the computational geometry application. If this were a C++ application, the GUI would have to be linked in, consuming link time, compile time, and memory footprint. In a Java application, the `.class` files for the `GUI` have to be deployed to the target platform.

Secondly, if a change to the `GraphicalApplication` causes the `Rectangle` to change for some reason, that change may force us to rebuild, retest, and redeploy the `ComputationalGeometryApplication`. If we forget to do this, that application may break in unpredictable ways.

A better design is to separate the two responsibilities into two completely different classes as shown in Figure 9-2. This design moves the computational portions of `Rectangle` into the `GeometricRectangle` class. Now changes made to the way rectangles are rendered cannot affect the `ComputationalGeometryApplication`.

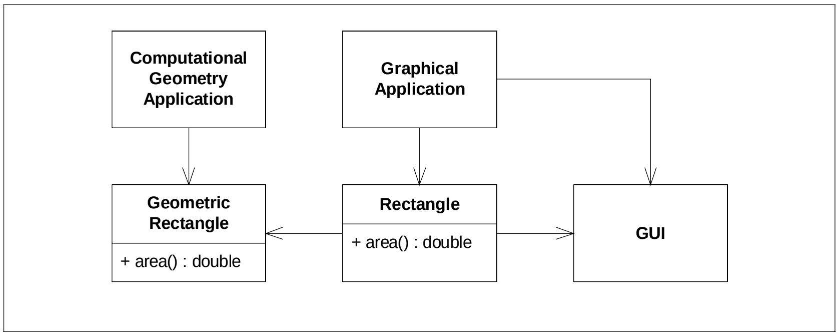

Figure 9-2
Separated Responsibilities

What is a Responsibility?

In the context of the Single Responsibility Principle (SRP) we define a responsibility to be "a reason for change." If you can think of more than one motive for changing a class, then that class has more than one responsibility. This is sometimes hard to see. We are accustomed to thinking of responsibility in groups. For example, consider the `Modem` interface in Listing 9-1. Most of us will agree that this interface looks perfectly reasonable. The four functions it declares are certainly functions belonging to a modem.

Listing 9-1
Modem.java -- SRP Violation

```
interface Modem
{
  public void dial(String pno);
  public void hangup();
  public void send(char c);
  public char recv();
}
```

However, there are two responsibilities being shown here. The first responsibility is connection management. The second is data communication. The `dial` and `hangup` functions manage the connection of the modem, while the `send` and `recv` functions communicate data.

Should these two responsibilities be separated? Almost certainly they should. The two sets of functions have almost nothing in common. They'll certainly change for different reasons. Moreover, they will be called from completely different parts of the applications that use them. Those different parts will change for different reasons as well.

Therefore the design in Figure 9-3 is probably better. It separates the two responsibilities into two separate interfaces[3]. This, at least, keeps the client applications from coupling the two responsibilities.

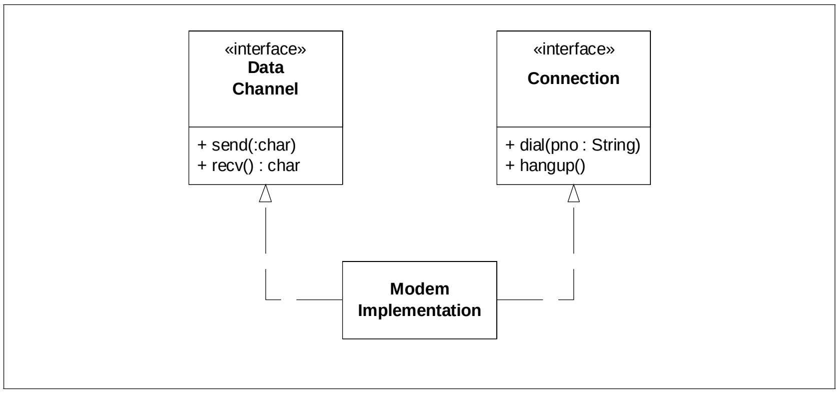

Figure 9-3
Separated Modem Interface

However, notice that I have recoupled the two responsibilities into a single `ModemImplementation` class. This is not desirable, but it may be necessary. There are often reasons, having to do with the details of the hardware or OS, that force us to couple things that we'd rather not couple. However, by separating their interfaces we have decoupled the concepts as far as the rest of the application is concerned.

We may view the `ModemImplementation` class is a kludge, or a wart; however, notice that all dependencies flow *away* from it. Nobody need depend upon this class. Nobody except `main` needs to know that it exists. Thus, we've put the ugly bit behind a fence. It's ugliness need not leak out and pollute the rest of the application.

Conclusion

The SRP is one of the simplest of the principle, and one of the hardest to get right. Conjoining responsibilities is something that we do naturally. Finding and separating those responsibilities from one another is much of what software design is really about. Indeed, the rest of the principles we will discuss come back to this issue in one way or another.

3. We'll see more of this in Chapter 13, when we study the Interface Segregation Principle (ISP).

Bibliography

[DeMarco79]: *Structured Analysis and System Specification,* Tom DeMarco, Yourdon Press Computing Series, 1979

[PageJones88]: *The Practical Guide to Structured Systems Design*, 2d. ed., Meilir Page-Jones, Yourdon Press Computing Series, 1988