Chapter 8. The Single-Responsibility Principle (SRP)

> *None but Buddha himself must take the responsibility of giving out occult secrets . . .*
>
> E. Cobham Brewer 18101897, *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*, which they defined as the functional relatedness of the elements of a module. In this chapter, we modify that meaning a bit and relate cohesion to the forces that cause a module, or a class, to change.

[1] [DeMarco79], p. 310

[2] [PageJones88], p. 82

The Single-Responsibility Principle

A class should have only one reason to change.

Consider the bowling game from Chapter 6. For most of its development, the `Game` class was handling two separate responsibilities: keeping track of the current frame and 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.

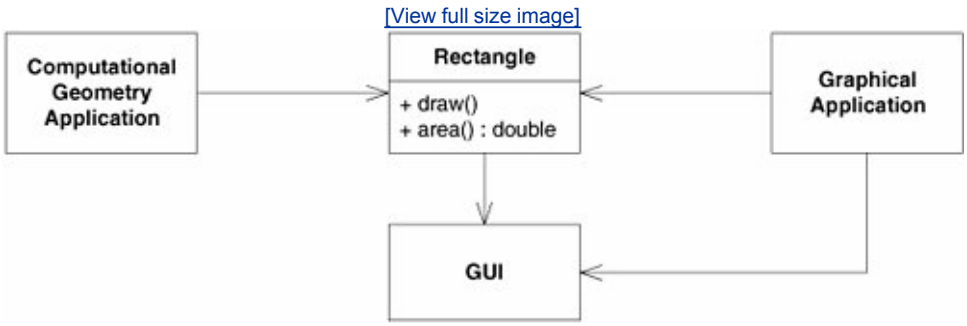
Why was it important to separate these two responsibilities into separate classes? The reason is that each responsibility is an axis of change. When the requirements change, that change will be manifest through a change in responsibility among the classes. If a class assumes more than one responsibility,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

If a class has more than one responsibility, the responsibilities become coupled. Changes to one responsibility may impair or inhibit the class's 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 8-1. The `Rectangle` class has two methods shown. One draws the rectangle on the screen, and the other computes the area of the rectangle.

Figure 8-1. More than one responsibility

[View full size image]



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

This design violates 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 GUI.

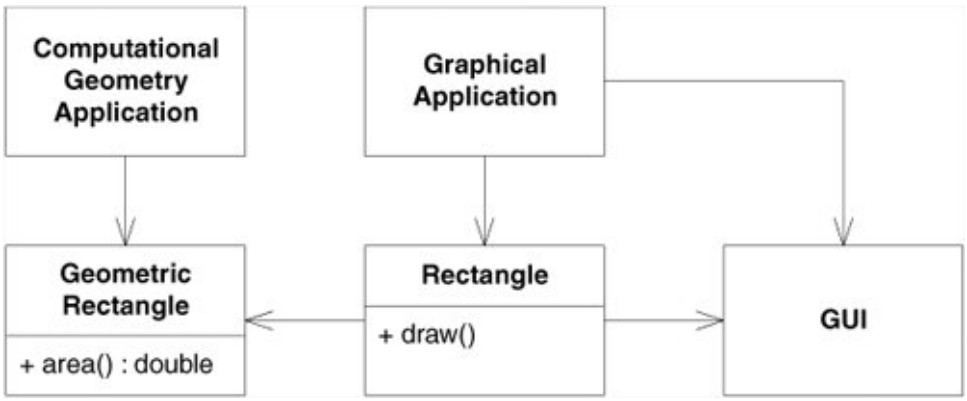
The violation of SRP causes several nasty problems. First, we must include `GUI` in the computational geometry application. In .NET, the GUI assembly would have to be built and deployed with the computational geometry application.

Second, 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 8-2. 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 8-2. Separated responsibilities



PREV NEXT

Defining a Responsibility

In the context of the SRP, we define a responsibility to be *a reason for change*. If you can think of more than one motive for changing a class, that class has more than one responsibility. This is sometimes difficult to see. We are accustomed to thinking of responsibility in groups. For example, consider the `Modem` interface in Listing 8-1. Most of us will agree that this interface looks perfectly reasonable. The four functions it declares are certainly functions belonging to a modem.

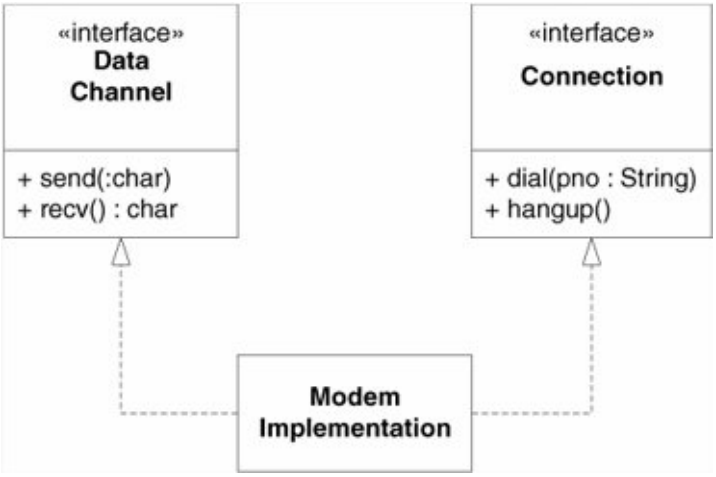
Listing 8-1. `Modem.cs -- SRP violation`

```
public interface Modem
{
  public void Dial(string pno);
  public void Hangup();
  public void Send(char c);
  public char Recv();
}
```

However, two responsibilities are 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; the `send` and `recv` functions communicate data.

Should these two responsibilities be separated? That depends on how the application is changing. If the application changes in ways that affect the signature of the connection functions, the design will smell of rigidity, because the classes that call `send` and `read` will have to be recompiled and redeployed more often than we like. In that case, the two responsibilities should be separated, as shown in Figure 8-3. This keeps the client applications from coupling the two responsibilities.

Figure 8-3. Separated modem interface



If, on the other hand, the application is not changing in ways that cause the two responsibilities to change at different times, there is no need to separate them. Indeed, separating them would smell of needless complexity.

There is a corrolary here. *An axis of change is an axis of change only if the changes occur*. It is not wise to apply SRPor any other principle, for that matterif there is no symptom.

PREV NEXT

Separating Coupled Responsibilities

Note that in Figure 8-3, I kept both responsibilities coupled in the `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 operating system, 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 as a kludge or a wart; however, note that all dependencies flow *away* from it. Nobody needs to depend on this class. Nobody except `main` needs to know that it exists. Thus, we've put the ugly bit behind a fence. Its ugliness need not leak out and pollute the rest of the application.

Persistence

Figure 8-4 shows a common violation of SRP. The `Employee` class contains business rules and persistence control. These two responsibilities should almost never be mixed. Business rules tend to change frequently, and although persistence may not change as frequently, it changes for completely different reasons. Binding business rules to the persistence subsystem is asking for trouble.

Figure 8-4. Coupled persistence

Fortunately, as we saw in Chapter 4, the practice of test-driven development will usually force these two responsibilities to be separated long before the design begins to smell. However, if the tests did not force the separation, and if the smells of rigidity and fragility become strong, the design should be refactored, using the FACADE, DAO (Data Access Object), or PROXY patterns to separate the two responsibilities.

Conclusion

The Single-Responsibility Principle is one of the simplest of the principles but one of the most difficult to get right. Conjoining responsibilities is something that we do naturally. Finding and separating those responsibilities is much of what software design is really about. Indeed, the rest of the principles we discuss come back to this issue in one way or another.

Bibliography

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

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