7 SRP: The Single Responsibility Principle

Of all the SOLID principles, the Single Responsibility Principle (SRP) might be the least well understood. That's likely because it has a particularly inappropriate name. It is too easy for programmers to hear the name and then assume that it means that every module should do just one thing.

Make no mistake, there *is* a principle like that. A *function* should do one, and only one, thing. We use that principle when we are refactoring large functions into smaller functions; we use it at the lowest levels. But it is not one of the SOLID principles—it is not the SRP.

Historically, the SRP has been described this way:

> *A module should have one, and only one, reason to change.*

Software systems are changed to satisfy users and stakeholders; those users and stakeholders *are* the "reason to change" that the principle is talking about. Indeed, we can rephrase the principle to say this:

> *A module should be responsible to one, and only one, user or stakeholder.*

Unfortunately, the words "user" and "stakeholder" aren't really the right words to use here. There will likely be more than one user or stakeholder who wants the system changed in the same way. Instead, we're really referring to a group—one or more people who require that change. We'll refer to that group as an *actor*.

Thus the final version of the SRP is:

> *A module should be responsible to one, and only one, actor.*

Now, what do we mean by the word "module"? The simplest definition is just a source file. Most of the time that definition works fine. Some languages and development environments, though, don't use source files to contain their code. In those cases a module is just a cohesive set of functions and data structures.

That word "cohesive" implies the SRP. Cohesion is the force that binds together the code responsible to a single actor.

Perhaps the best way to understand this principle is by looking at the symptoms of violating it.

SYMPTOM 1: ACCIDENTAL DUPLICATION

My favorite example is the `Employee` class from a payroll application. It has three methods: `calculatePay()`, `reportHours()`, and `save()` (Figure 7.1).

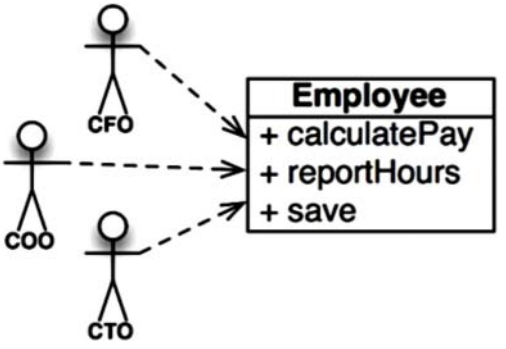

Figure 7.1 The `Employee` class

This class violates the SRP because those three methods are responsible to three very different actors.

- The `calculatePay()` method is specified by the accounting department, which reports to the CFO.
- The `reportHours()` method is specified and used by the human resources department, which reports to the COO.
- The `save()` method is specified by the database administrators (DBAs), who report to the CTO.

By putting the source code for these three methods into a single `Employee` class, the developers have coupled each of these actors to the others. This

coupling can cause the actions of the CFO's team to affect something that the COO's team depends on.

For example, suppose that the `calculatePay()` function and the `reportHours()` function share a common algorithm for calculating non-overtime hours. Suppose also that the developers, who are careful not to duplicate code, put that algorithm into a function named `regularHours()` (Figure 7.2).

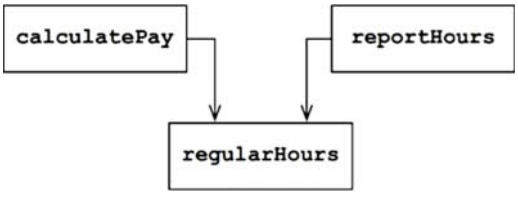

Figure 7.2 Shared algorithm

Now suppose that the CFO's team decides that the way non-overtime hours are calculated needs to be tweaked. In contrast, the COO's team in HR does not want that particular tweak because they use non-overtime hours for a different purpose.

A developer is tasked to make the change, and sees the convenient `regularHours()` function called by the `calculatePay()` method. Unfortunately, that developer does not notice that the function is also called by the `reportHours()` function.

The developer makes the required change and carefully tests it. The CFO's team validates that the new function works as desired, and the system is deployed.

Of course, the COO's team doesn't know that this is happening. The HR personnel continue to use the reports generated by the `reportHours()` function—but now they contain incorrect numbers. Eventually the problem is discovered, and the COO is livid because the bad data has cost his budget millions of dollars.

We've all seen things like this happen. These problems occur because we put code that different actors depend on into close proximity. The SRP says to *separate the code that different actors depend on*.

SYMPTOM 2: MERGES

It's not hard to imagine that merges will be common in source files that contain many different methods. This situation is especially likely if those methods are responsible to different actors.

For example, suppose that the CTO's team of DBAs decides that there should be a simple schema change to the `Employee` table of the database. Suppose also that the COO's team of HR clerks decides that they need a change in the format of the hours report.

Two different developers, possibly from two different teams, check out the `Employee` class and begin to make changes. Unfortunately their changes collide. The result is a merge.

I probably don't need to tell you that merges are risky affairs. Our tools are pretty good nowadays, but no tool can deal with every merge case. In the end, there is always risk.

In our example, the merge puts both the CTO and the COO at risk. It's not inconceivable that the CFO could be affected as well.

There are many other symptoms that we could investigate, but they all involve multiple people changing the same source file for different reasons.

Once again, the way to avoid this problem is to *separate code that supports different actors*.

SOLUTIONS

There are many different solutions to this problem. Each moves the functions into different classes.

Perhaps the most obvious way to solve the problem is to separate the data from the functions. The three classes share access to `EmployeeData`, which is a simple data structure with no methods (Figure 7.3). Each class holds only the source code necessary for its particular function. The three classes are not allowed to know about each other. Thus any accidental duplication is avoided.

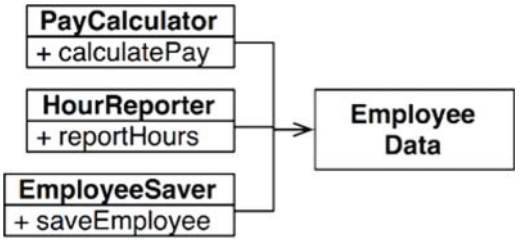

Figure 7.3 The three classes do not know about each other

The downside of this solution is that the developers now have three classes that they have to instantiate and track. A common solution to this dilemma is to use the *Facade* pattern (Figure 7.4).

Figure 7.4 The `Facade` pattern

The `EmployeeFacade` contains very little code. It is responsible for instantiating and delegating to the classes with the functions.

Some developers prefer to keep the most important business rules closer to the data. This can be done by keeping the most important method in the original `Employee` class and then using that class as a *Facade* for the lesser functions (Figure 7.5).

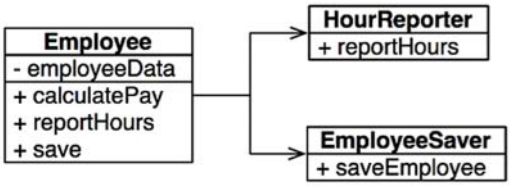

Figure 7.5 The most important method is kept in the original `Employee` class and used as a *Facade* for the lesser functions

You might object to these solutions on the basis that every class would contain just one function. This is hardly the case. The number of functions required to calculate pay, generate a report, or save the data is likely to be large in each case. Each of those classes would have many *private* methods in them.

Each of the classes that contain such a family of methods is a scope. Outside of that scope, no one knows that the private members of the family exist.

CONCLUSION

The Single Responsibility Principle is about functions and classes—but it reappears in a different form at two more levels. At the level of components, it becomes the Common Closure Principle. At the architectural level, it becomes the Axis of Change responsible for the creation of Architectural Boundaries. We'll be studying all of these ideas in the chapters to come.