Chapter 76. The Single Responsibility Principle

Robert C. Martin (Uncle Bob)

ONE OF THE MOST FOUNDATIONAL PRINCIPLES OF GOOD DESIGN IS:

> *Gather together those things that change for the same reason, and separate those things that change for different reasons.*

This principle is often known as the *single responsibility principle*, or SRP. In short, it says that a subsystem, module, class, or even a function, should not have more than one reason to change. The classic example is a class that has methods that deal with business rules, reports, and databases:

```
public class Employee {
    public Money calculatePay() ...
    public String reportHours() ...
    public void save() ...
}
```

Some programmers might think that putting these three functions together in the same class is perfectly appropriate. After all, classes are supposed to be collections of functions that operate on common variables. However, the problem is that the three functions change for entirely different reasons. The `calculatePay` function will change whenever the business rules for calculating pay do. The `reportHours` function will change whenever someone wants a different format for the report. The `save` function will change whenever the DBAs change the database schema. These three reasons to change combine to make `Employee` very volatile. It will change for *any* of those reasons. More importantly, any classes that depend upon `Employee` will be affected by those changes.

Good system design means that we separate the system into components that can be independently deployed. Independent deployment means that if we change one component, we do not have to redeploy any of the others. However, if `Employee` is used heavily by many other classes in other components, then every change to `Employee` is likely to cause the other components to be redeployed, thus negating a major benefit of component design (or SOA, if you prefer the trendier name). The following simple partitioning resolves the issues:

```
public class Employee {
    public Money calculatePay() ...
}
public class EmployeeReporter {
    public String reportHours(Employee e) ...
}
public class EmployeeRepository {
    public void save(Employee e) ...
}
```

```
}
```

Each class can be placed in a component of its own. Or rather, all the reporting classes can go into the reporting component. All the database-related classes can go into the repository component. And all the business rules can go into the business rule component.

The astute reader will see that there are still dependencies in the above solution. That `Employee` is still depended upon by the other classes. So if `Employee` is modified, the other classes will likely have to be recompiled and redeployed. Thus, `Employee` cannot be modified and then independently deployed. However, the other classes can be modified and independently deployed. No modification of one of them can force any of the others to be recompiled or redeployed. Even `Employee` could be independently deployed through a careful use of the *dependency inversion principle* (DIP), but that's a topic for a different book.[9]

Careful application of the SRP, separating things that change for different reasons, is one of the keys to creating designs that have an independently deployable component structure.