8 OCP: The Open-Closed Principle

The Open-Closed Principle (OCP) was coined in 1988 by Bertrand Meyer.[1] It says:

> *A software artifact should be open for extension but closed for modification.*

In other words, the behavior of a software artifact ought to be extendible, without having to modify that artifact.

This, of course, is the most fundamental reason that we study software architecture. Clearly, if simple extensions to the requirements force massive changes to the software, then the architects of that software system have engaged in a spectacular failure.

Most students of software design recognize the OCP as a principle that guides them in the design of classes and modules. But the principle takes on even greater significance when we consider the level of architectural components.

A thought experiment will make this clear.

A THOUGHT EXPERIMENT

Imagine, for a moment, that we have a system that displays a financial summary on a web page. The data on the page is scrollable, and negative numbers are rendered in red.

Now imagine that the stakeholders ask that this same information be turned into a report to be printed on a black-and-white printer. The report should be properly paginated, with appropriate page headers, page footers, and column labels. Negative numbers should be surrounded by parentheses.

Clearly, some new code must be written. But how much old code will have to change?

1. Bertrand Meyer. *Object Oriented Software Construction*, Prentice Hall, 1988, p. 23.

A good software architecture would reduce the amount of changed code to the barest minimum. Ideally, zero.

How? By properly separating the things that change for different reasons (the Single Responsibility Principle), and then organizing the dependencies between those things properly (the Dependency Inversion Principle).

By applying the SRP, we might come up with the data-flow view shown in Figure 8.1. Some analysis procedure inspects the financial data and produces reportable data, which is then formatted appropriately by the two reporter processes.

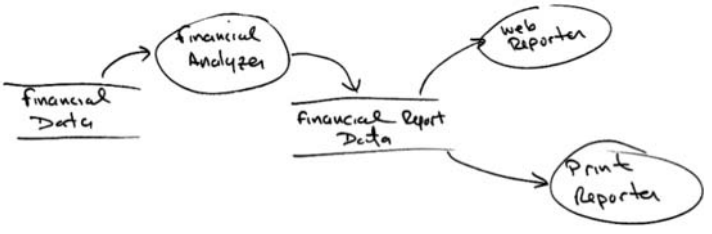

Figure 8.1 Applying the SRP

The essential insight here is that generating the report involves two separate responsibilities: the calculation of the reported data, and the presentation of that data into a web- and printer-friendly form.

Having made this separation, we need to organize the source code dependencies to ensure that changes to one of those responsibilities do not cause changes in the other. Also, the new organization should ensure that the behavior can be extended without undo modification.

We accomplish this by partitioning the processes into classes, and separating those classes into components, as shown by the double lines in the diagram in Figure 8.2. In this figure, the component at the upper left is the *Controller*. At the upper right, we have the *Interactor*. At the lower right, there is the *Database*. Finally, at the lower left, there are four components that represent the *Presenters* and the *Views*.

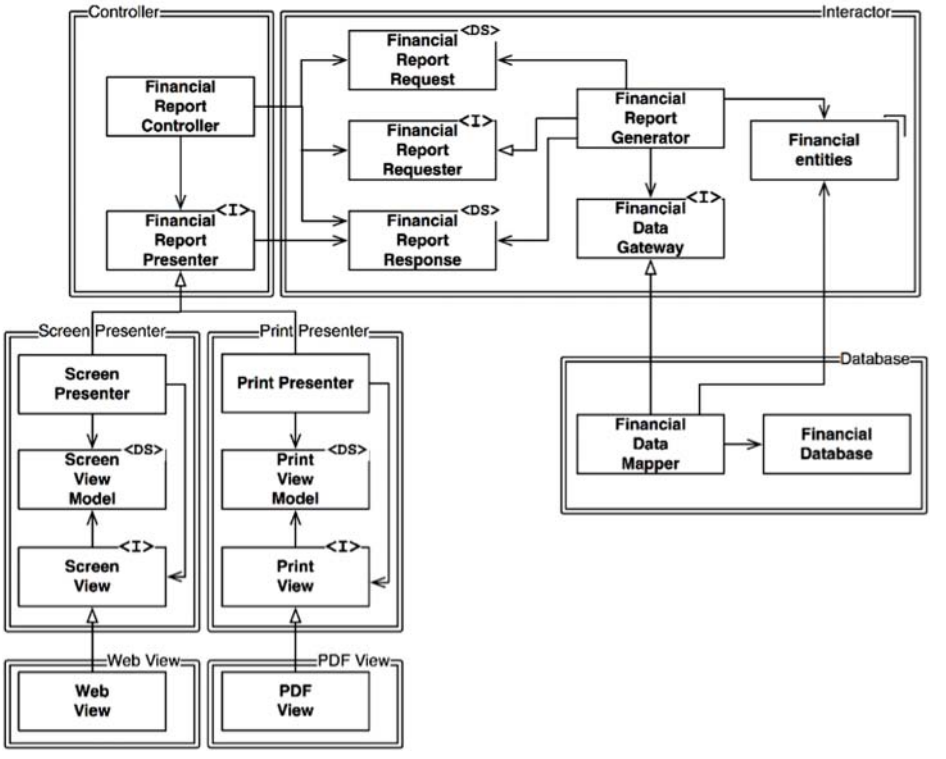

Figure 8.2 Partitioning the processes into classes and separating the classes into components

Classes marked with <I> are interfaces; those marked with <DS> are data structures. Open arrowheads are *using* relationships. Closed arrowheads are *implements* or *inheritance* relationships.

The first thing to notice is that all the dependencies are *source code* dependencies. An arrow pointing from class A to class B means that the source code of class A mentions the name of class B, but class B mentions nothing about class A. Thus, in Figure 8.2, `FinancialDataMapper` knows about `FinancialDataGateway` through an *implements* relationship, but `FinancialGateway` knows nothing at all about `FinancialDataMapper`.

The next thing to notice is that each double line is crossed *in one direction only*. This means that all component relationships are unidirectional, as

shown in the component graph in Figure 8.3. These arrows point toward the components that we want to protect from change.

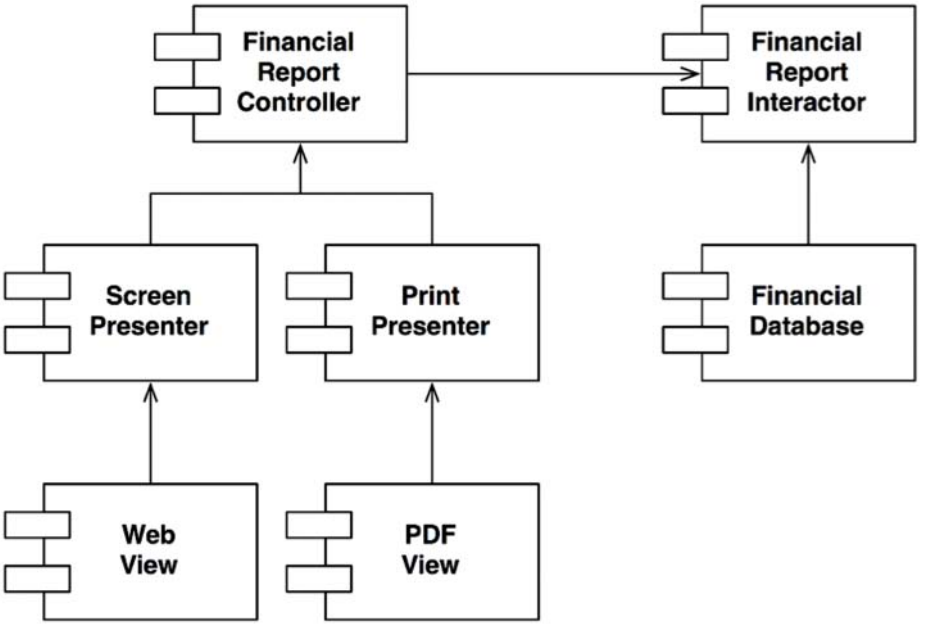

Figure 8.3 The component relationships are unidirectional

Let me say that again: If component A should be protected from changes in component B, then component B should depend on component A.

We want to protect the *Controller* from changes in the *Presenters*. We want to protect the *Presenters* from changes in the *Views*. We want to protect the *Interactor* from changes in—well, *anything*.

The *Interactor* is in the position that best conforms to the OCP. Changes to the *Database*, or the *Controller*, or the *Presenters*, or the *Views*, will have no impact on the *Interactor*.

Why should the *Interactor* hold such a privileged position? Because it contains the business rules. The *Interactor* contains the highest-level policies

of the application. All the other components are dealing with peripheral concerns. The *Interactor* deals with the central concern.

Even though the *Controller* is peripheral to the *Interactor*, it is nevertheless central to the *Presenters* and *Views*. And while the *Presenters* might be peripheral to the *Controller*, they are central to the *Views*.

Notice how this creates a hierarchy of protection based on the notion of "level." *Interactors* are the highest-level concept, so they are the most protected. *Views* are among the lowest-level concepts, so they are the least protected. *Presenters* are higher level than *Views*, but lower level than the *Controller* or the *Interactor*.

This is how the OCP works at the architectural level. Architects separate functionality based on how, why, and when it changes, and then organize that separated functionality into a hierarchy of components. Higher-level components in that hierarchy are protected from the changes made to lower-level components.

DIRECTIONAL CONTROL

If you recoiled in horror from the class design shown earlier, look again. Much of the complexity in that diagram was intended to make sure that the dependencies between the components pointed in the correct direction.

For example, the `FinancialDataGateway` interface between the `FinancialReportGenerator` and the `FinancialDataMapper` exists to invert the dependency that would otherwise have pointed from the *Interactor* component to the *Database* component. The same is true of the `FinancialReportPresenter` interface, and the two *View* interfaces.

INFORMATION HIDING

The `FinancialReportRequester` interface serves a different purpose. It is there to protect the `FinancialReportController` from knowing too much

about the internals of the *Interactor*. If that interface were not there, then the *Controller* would have transitive dependencies on the `FinancialEntities`.

Transitive dependencies are a violation of the general principle that software entities should not depend on things they don't directly use. We'll encounter that principle again when we talk about the Interface Segregation Principle and the Common Reuse Principle.

So, even though our first priority is to protect the *Interactor* from changes to the *Controller*, we also want to protect the *Controller* from changes to the *Interactor* by hiding the internals of the *Interactor*.

CONCLUSION

The OCP is one of the driving forces behind the architecture of systems. The goal is to make the system easy to extend without incurring a high impact of change. This goal is accomplished by partitioning the system into components, and arranging those components into a dependency hierarchy that protects higher-level components from changes in lower-level components.