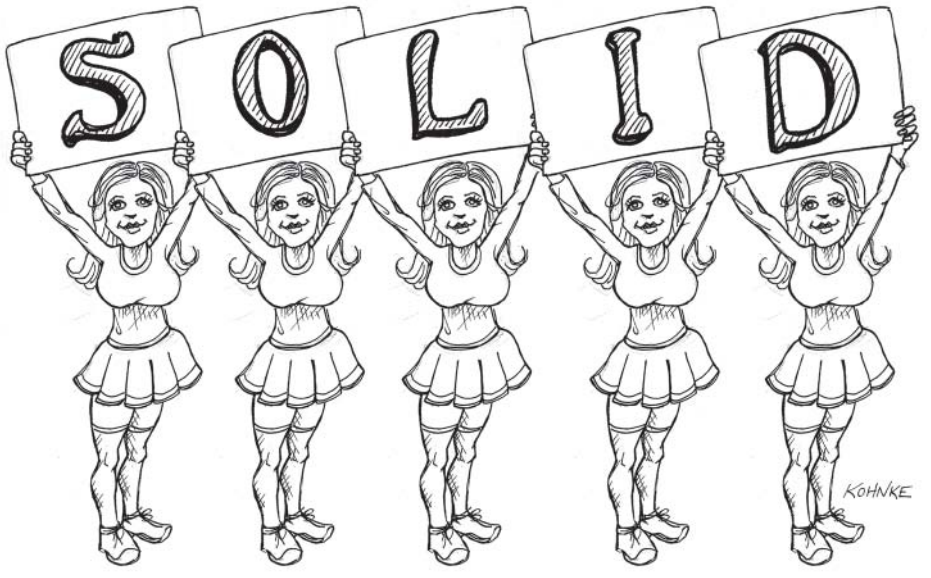
III
Design Principles



Good software systems begin with clean code. On the one hand, if the bricks aren't well made, the architecture of the building doesn't matter much. On the other hand, you can make a substantial mess with well-made bricks. This is where the SOLID principles come in.

The SOLID principles tell us how to arrange our functions and data structures into classes, and how those classes should be interconnected. The use of the word "class" does not imply that these principles are applicable only to object-oriented software. A class is simply a coupled grouping of functions and data. Every software system has such groupings, whether they are called classes or not. The SOLID principles apply to those groupings.

The goal of the principles is the creation of mid-level software structures that:

- Tolerate change,
- Are easy to understand, and
- Are the basis of components that can be used in many software systems.

The term "mid-level" refers to the fact that these principles are applied by programmers working at the module level. They are applied just above the level of the code and help to define the kinds of software structures used within modules and components.

Just as it is possible to create a substantial mess with well-made bricks, so it is also possible to create a system-wide mess with well-designed mid-level components. For this reason, once we have covered the SOLID principles, we will move on to their counterparts in the component world, and then to the principles of high-level architecture.

The history of the SOLID principles is long. I began to assemble them in the late 1980s while debating software design principles with others on USENET (an early kind of Facebook). Over the years, the principles have shifted and changed. Some were deleted. Others were merged. Still others were added. The final grouping stabilized in the early 2000s, although I presented them in a different order.

In 2004 or thereabouts, Michael Feathers sent me an email saying that if I rearranged the principles, their first words would spell the word SOLID—and thus the SOLID principles were born.

The chapters that follow describe each principle more thoroughly. Here is the executive summary:

- **SRP:** The Single Responsibility Principle

  An active corollary to Conway's law: The best structure for a software system is heavily influenced by the social structure of the organization that uses it so that each software module has one, and only one, reason to change.
- **OCP:** The Open-Closed Principle

  Bertrand Meyer made this principle famous in the 1980s. The gist is that for software systems to be easy to change, they must be designed to allow the behavior of those systems to be changed by adding new code, rather than changing existing code.
- **LSP:** The Liskov Substitution Principle

  Barbara Liskov's famous definition of subtypes, from 1988. In short, this principle says that to build software systems from interchangeable parts, those parts must adhere to a contract that allows those parts to be substituted one for another.
- **ISP:** The Interface Segregation Principle

  This principle advises software designers to avoid depending on things that they don't use.
- **DIP:** The Dependency Inversion Principle

  The code that implements high-level policy should not depend on the code that implements low-level details. Rather, details should depend on policies.

These principles have been described in detail in many different publications[1] over the years. The chapters that follow will focus on the architectural implications of these principles instead of repeating those detailed discussions. If you are not already familiar with these principles, what follows is insufficient to understand them in detail and you would be well advised to study them in the footnoted documents.

1. For example, *Agile Software Development, Principles, Patterns, and Practices*, Robert C. Martin, Prentice Hall, 2002, http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod, and https://en.wikipedia.org/wiki/SOLID_(object-oriented_design) (or just google SOLID).

This page intentionally left blank

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).

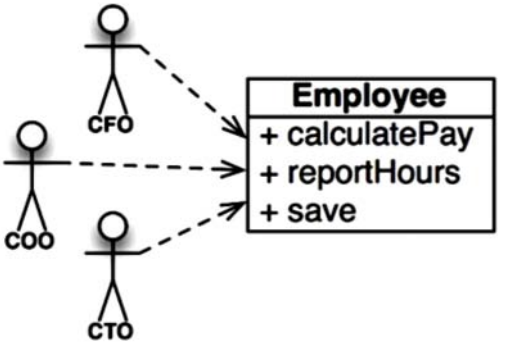


Figure 7.1 The `Employee` class

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).

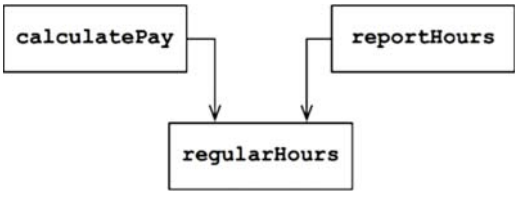


Figure 7.2 Shared algorithm

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.

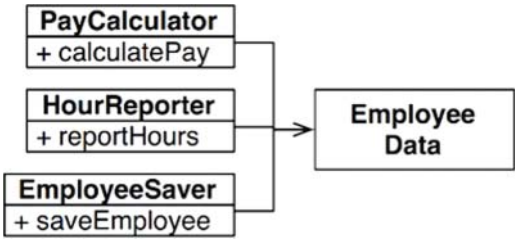


Figure 7.3 The three classes do not know about each other

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).

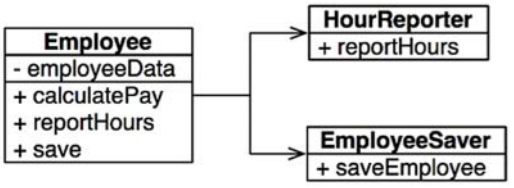


Figure 7.5 The most important method is kept in the original `Employee` class and used as a *Facade* for the lesser functions

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.

This page intentionally left blank

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.

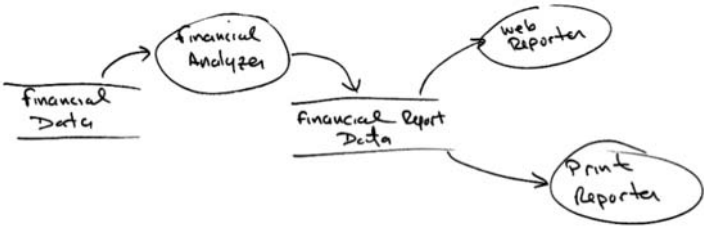


Figure 8.1 Applying the SRP

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*.

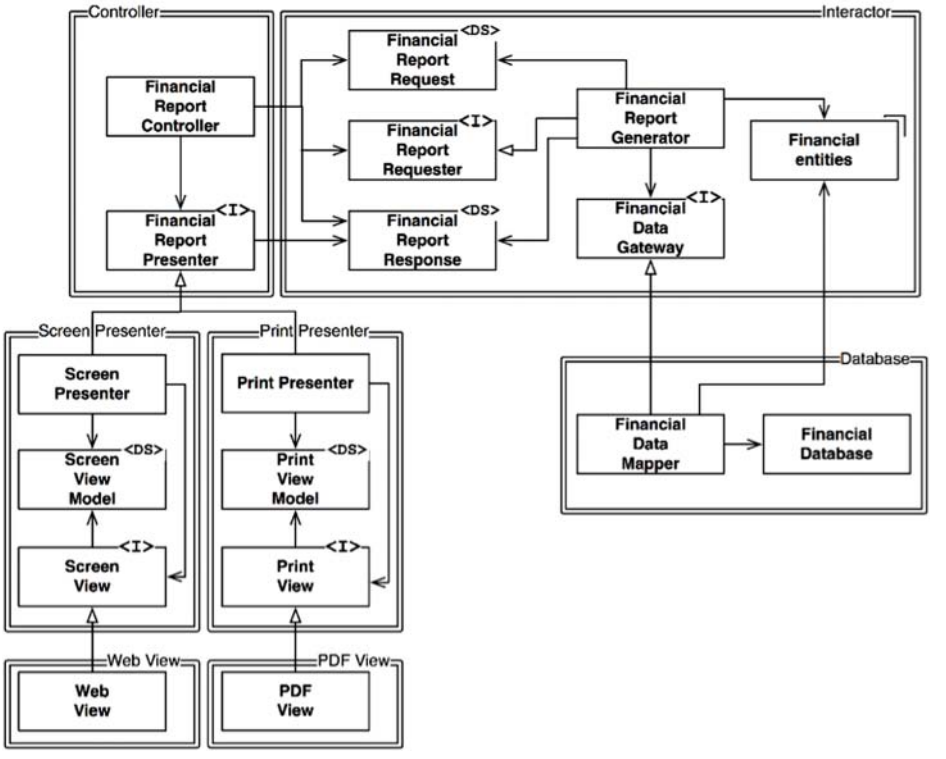


Figure 8.2 Partitioning the processes into classes and separating the classes into components

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.

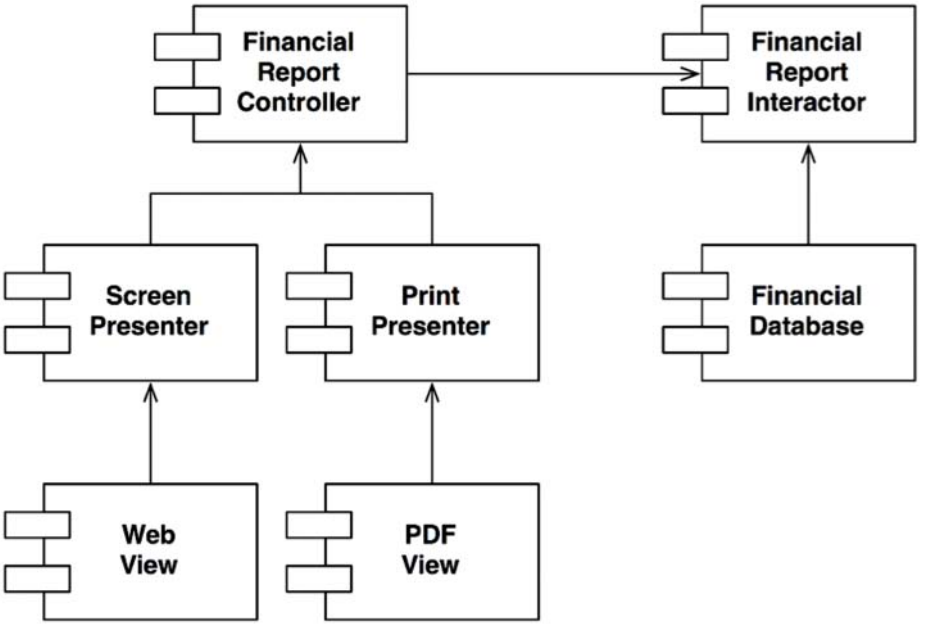


Figure 8.3 The component relationships are unidirectional

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.

This page intentionally left blank

9 LSP: The Liskov Substitution Principle

In 1988, Barbara Liskov wrote the following as a way of defining subtypes.

> *What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.*[1]

To understand this idea, which is known as the Liskov Substitution Principle (LSP), let's look at some examples.

GUIDING THE USE OF INHERITANCE

Imagine that we have a class named `License`, as shown in Figure 9.1. This class has a method named `calcFee()`, which is called by the `Billing` application. There are two "subtypes" of `License`: `PersonalLicense` and `BusinessLicense`. They use different algorithms to calculate the license fee.

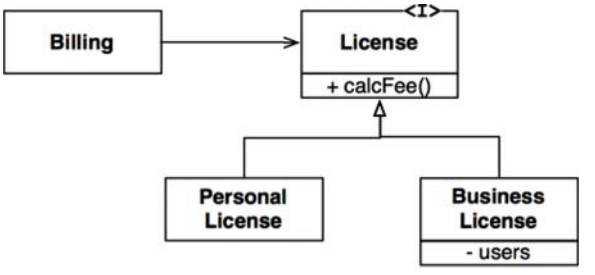


Figure 9.1 `License`, and its derivatives, conform to LSP

This design conforms to the LSP because the behavior of the `Billing` application does not depend, in any way, on which of the two subtypes it uses. Both of the subtypes are substitutable for the `License` type.

1. Barbara Liskov, "Data Abstraction and Hierarchy," *SIGPLAN Notices* 23, 5 (May 1988).

THE SQUARE/RECTANGLE PROBLEM

The canonical example of a violation of the LSP is the famed (or infamous, depending on your perspective) square/rectangle problem (Figure 9.2).

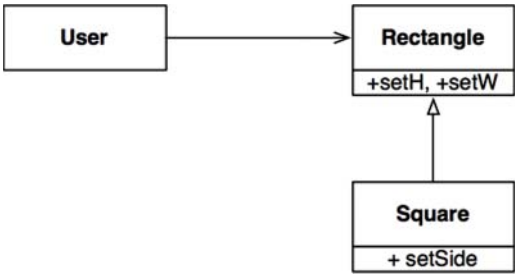


Figure 9.2 The infamous square/rectangle problem

In this example, `Square` is not a proper subtype of `Rectangle` because the height and width of the `Rectangle` are independently mutable; in contrast, the height and width of the `Square` must change together. Since the `User` believes it is communicating with a `Rectangle`, it could easily get confused. The following code shows why:

```
Rectangle r = …
r.setW(5);
r.setH(2);
assert(r.area() == 10);
```

If the … code produced a `Square`, then the assertion would fail.

The only way to defend against this kind of LSP violation is to add mechanisms to the `User` (such as an `if` statement) that detects whether the `Rectangle` is, in fact, a `Square`. Since the behavior of the `User` depends on the types it uses, those types are not substitutable.

LSP AND ARCHITECTURE

In the early years of the object-oriented revolution, we thought of the LSP as a way to guide the use of inheritance, as shown in the previous sections. However, over the years the LSP has morphed into a broader principle of software design that pertains to interfaces and implementations.

The interfaces in question can be of many forms. We might have a Java-style interface, implemented by several classes. Or we might have several Ruby classes that share the same method signatures. Or we might have a set of services that all respond to the same REST interface.

In all of these situations, and more, the LSP is applicable because there are users who depend on well-defined interfaces, and on the substitutability of the implementations of those interfaces.

The best way to understand the LSP from an architectural viewpoint is to look at what happens to the architecture of a system when the principle is violated.

EXAMPLE LSP VIOLATION

Assume that we are building an aggregator for many taxi dispatch services. Customers use our website to find the most appropriate taxi to use, regardless of taxi company. Once the customer makes a decision, our system dispatches the chosen taxi by using a restful service.

Now assume that the URI for the restful dispatch service is part of the information contained in the driver database. Once our system has chosen a driver appropriate for the customer, it gets that URI from the driver record and then uses it to dispatch the driver.

Suppose Driver Bob has a dispatch URI that looks like this:

```
purplecab.com/driver/Bob
```

Our system will append the dispatch information onto this URI and send it with a PUT, as follows:

```
purplecab.com/driver/Bob
        /pickupAddress/24 Maple St.
        /pickupTime/153
        /destination/ORD
```

Clearly, this means that all the dispatch services, for all the different companies, must conform to the same REST interface. They must treat the `pickupAddress`, `pickupTime`, and `destination` fields identically.

Now suppose the Acme taxi company hired some programmers who didn't read the spec very carefully. They abbreviated the destination field to just `dest`. Acme is the largest taxi company in our area, and Acme's CEO's ex-wife is our CEO's new wife, and … Well, you get the picture. What would happen to the architecture of our system?

Obviously, we would need to add a special case. The dispatch request for any Acme driver would have to be constructed using a different set of rules from all the other drivers.

The simplest way to accomplish this goal would be to add an `if` statement to the module that constructed the dispatch command:

```
if (driver.getDispatchUri().startsWith("acme.com"))…
```

But, of course, no architect worth his or her salt would allow such a construction to exist in the system. Putting the word "acme" into the code itself creates an opportunity for all kinds of horrible and mysterious errors, not to mention security breaches.

For example, what if Acme became even more successful and bought the Purple Taxi company. What if the merged company maintained the separate

brands and the separate websites, but unified all of the original companies' systems? Would we have to add another if statement for "purple"?

Our architect would have to insulate the system from bugs like this by creating some kind of dispatch command creation module that was driven by a configuration database keyed by the dispatch URI. The configuration data might look something like this:

URI	Dispatch Format
`Acme.com`	`/pickupAddress/%s/pickupTime/%s/dest/%s`
`*.*`	`/pickupAddress/%s/pickupTime/%s/destination/%s`

And so our architect has had to add a significant and complex mechanism to deal with the fact that the interfaces of the restful services are not all substitutable.

CONCLUSION

The LSP can, and should, be extended to the level of architecture. A simple violation of substitutability, can cause a system's architecture to be polluted with a significant amount of extra mechanisms.

10 ISP: The Interface Segregation Principle

The Interface Segregation Principle (ISP) derives its name from the diagram shown in Figure 10.1.

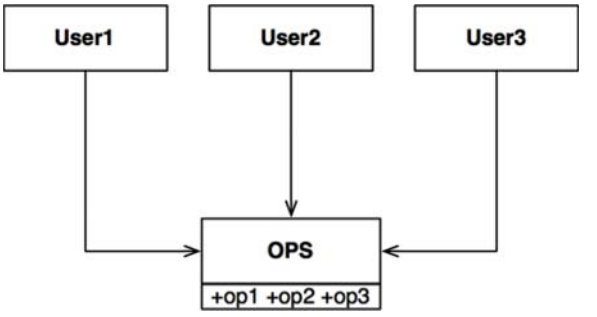


Figure 10.1 The Interface Segregation Principle

In the situation illustrated in Figure 10.1, there are several users who use the operations of the `OPS` class. Let's assume that `User1` uses only `op1`, `User2` uses only `op2`, and `User3` uses only `op3`.

Now imagine that `OPS` is a class written in a language like Java. Clearly, in that case, the source code of `User1` will inadvertently depend on `op2` and `op3`, even though it doesn't call them. This dependence means that a change to the source code of `op2` in `OPS` will force `User1` to be recompiled and redeployed, even though nothing that it cared about has actually changed.

This problem can be resolved by segregating the operations into interfaces as shown in Figure 10.2.

Again, if we imagine that this is implemented in a statically typed language like Java, then the source code of `User1` will depend on `U1Ops`, and `op1`, but will not depend on `OPS`. Thus a change to `OPS` that `User1` does not care about will not cause `User1` to be recompiled and redeployed.

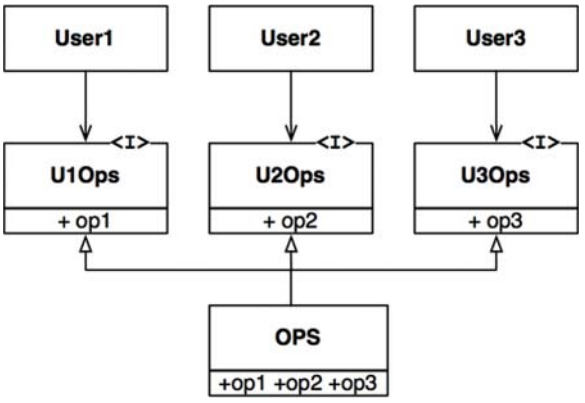


Figure 10.2 Segregated operations

ISP and Language

Clearly, the previously given description depends critically on language type. Statically typed languages like Java force programmers to create declarations that users must `import`, or `use`, or otherwise `include`. It is these `included` declarations in source code that create the source code dependencies that force recompilation and redeployment.

In dynamically typed languages like Ruby and Python, such declarations don't exist in source code. Instead, they are inferred at runtime. Thus there are no source code dependencies to force recompilation and redeployment. This is the primary reason that dynamically typed languages create systems that are more flexible and less tightly coupled than statically typed languages.

This fact could lead you to conclude that the ISP is a language issue, rather than an architecture issue.

ISP AND ARCHITECTURE

If you take a step back and look at the root motivations of the ISP, you can see a deeper concern lurking there. In general, it is harmful to depend on modules that contain more than you need. This is obviously true for source code dependencies that can force unnecessary recompilation and redeployment—but it is also true at a much higher, architectural level.

Consider, for example, an architect working on a system, S. He wants to include a certain framework, F, into the system. Now suppose that the authors of F have bound it to a particular database, D. So S depends on F. which depends on D (Figure 10.3).

Figure 10.3 A problematic architecture

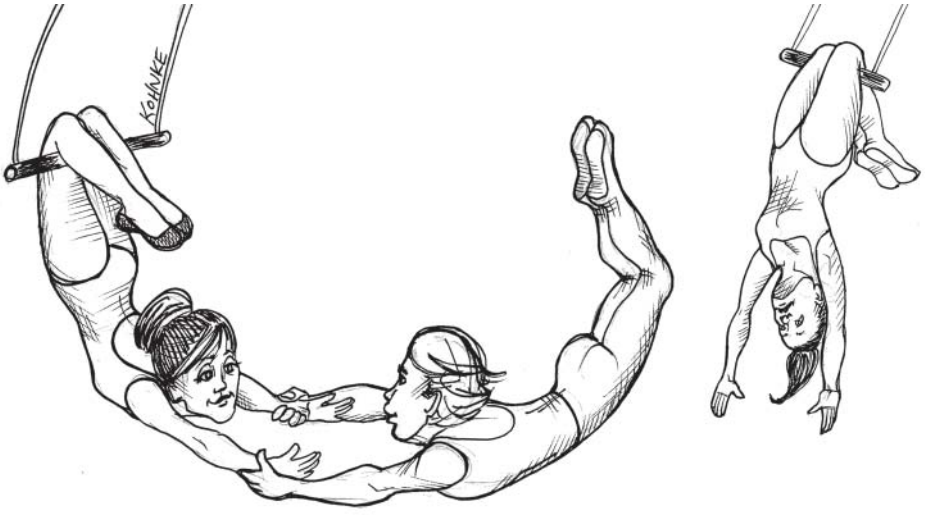
Now suppose that D contains features that F does not use and, therefore, that S does not care about. Changes to those features within D may well force the redeployment of F and, therefore, the redeployment of S. Even worse, a failure of one of the features within D may cause failures in F and S.

CONCLUSION

The lesson here is that depending on something that carries baggage that you don't need can cause you troubles that you didn't expect.

We'll explore this idea in more detail when we discuss the Common Reuse Principle in Chapter 13, "Component Cohesion."

11 DIP: The Dependency Inversion Principle



The Dependency Inversion Principle (DIP) tells us that the most flexible systems are those in which source code dependencies refer only to abstractions, not to concretions.

In a statically typed language, like Java, this means that the `use`, `import`, and `include` statements should refer only to source modules containing interfaces, abstract classes, or some other kind of abstract declaration. Nothing concrete should be depended on.

The same rule applies for dynamically typed languages, like Ruby and Python. Source code dependencies should not refer to concrete modules. However, in these languages it is a bit harder to define what a concrete module is. In particular, it is any module in which the functions being called are implemented.

Clearly, treating this idea as a rule is unrealistic, because software systems must depend on many concrete facilities. For example, the `String` class in Java is concrete, and it would be unrealistic to try to force it to be abstract. The source code dependency on the concrete `java.lang.string` cannot, and should not, be avoided.

By comparison, the `String` class is very stable. Changes to that class are very rare and tightly controlled. Programmers and architects do not have to worry about frequent and capricious changes to `String`.

For these reasons, we tend to ignore the stable background of operating system and platform facilities when it comes to DIP. We tolerate those concrete dependencies because we know we can rely on them not to change.

It is the *volatile* concrete elements of our system that we want to avoid depending on. Those are the modules that we are actively developing, and that are undergoing frequent change.

STABLE ABSTRACTIONS

Every change to an abstract interface corresponds to a change to its concrete implementations. Conversely, changes to concrete implementations do not always, or even usually, require changes to the interfaces that they implement. Therefore interfaces are less volatile than implementations.

Indeed, good software designers and architects work hard to reduce the volatility of interfaces. They try to find ways to add functionality to implementations without making changes to the interfaces. This is Software Design 101.

The implication, then, is that stable software architectures are those that avoid depending on volatile concretions, and that favor the use of stable abstract interfaces. This implication boils down to a set of very specific coding practices:

- **Don't refer to volatile concrete classes.** Refer to abstract interfaces instead. This rule applies in all languages, whether statically or dynamically typed. It also puts severe constraints on the creation of objects and generally enforces the use of *Abstract Factories*.
- **Don't derive from volatile concrete classes.** This is a corollary to the previous rule, but it bears special mention. In statically typed languages, inheritance is the strongest, and most rigid, of all the source code relationships; consequently, it should be used with great care. In dynamically typed languages, inheritance is less of a problem, but it is still a dependency—and caution is always the wisest choice.
- **Don't override concrete functions.** Concrete functions often require source code dependencies. When you override those functions, you do not eliminate those dependencies—indeed, you *inherit* them. To manage those dependencies, you should make the function abstract and create multiple implementations.
- **Never mention the name of anything concrete and volatile.** This is really just a restatement of the principle itself.

FACTORIES

To comply with these rules, the creation of volatile concrete objects requires special handling. This caution is warranted because, in virtually all languages, the creation of an object requires a source code dependency on the concrete definition of that object.

In most object-oriented languages, such as Java, we would use an *Abstract Factory* to manage this undesirable dependency.

The diagram in Figure 11.1 shows the structure. The `Application` uses the `ConcreteImpl` through the `Service` interface. However, the `Application`

must somehow create instances of the `ConcreteImpl`. To achieve this without creating a source code dependency on the `ConcreteImpl`, the `Application` calls the `makeSvc` method of the `ServiceFactory` interface. This method is implemented by the `ServiceFactoryImpl` class, which derives from `ServiceFactory`. That implementation instantiates the `ConcreteImpl` and returns it as a `Service`.

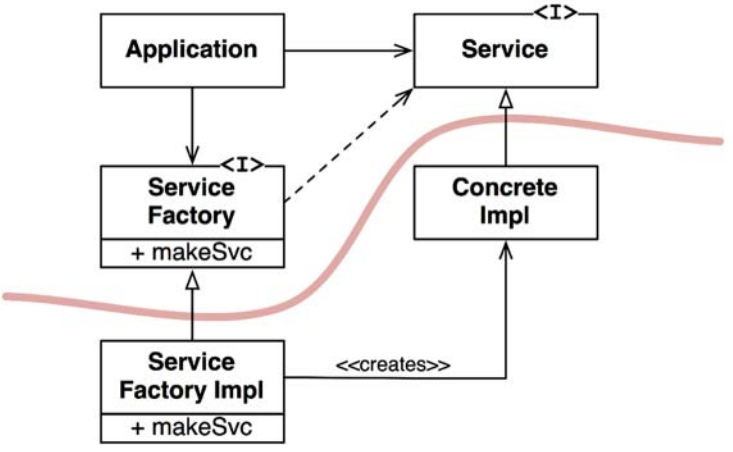


Figure 11.1 Use of the *Abstract Factory* pattern to manage the dependency

The curved line in Figure 11.1 is an architectural boundary. It separates the abstract from the concrete. All source code dependencies cross that curved line pointing in the same direction, toward the abstract side.

The curved line divides the system into two components: one abstract and the other concrete. The abstract component contains all the high-level business rules of the application. The concrete component contains all the implementation details that those business rules manipulate.

Note that the flow of control crosses the curved line in the opposite direction of the source code dependencies. The source code dependencies are inverted against the flow of control—which is why we refer to this principle as Dependency Inversion.

CONCRETE COMPONENTS

The concrete component in Figure 11.1 contains a single dependency, so it violates the DIP. This is typical. DIP violations cannot be entirely removed, but they can be gathered into a small number of concrete components and kept separate from the rest of the system.

Most systems will contain at least one such concrete component—often called `main` because it contains the `main`[1] function. In the case illustrated in Figure 11.1, the `main` function would instantiate the `ServiceFactoryImpl` and place that instance in a global variable of type `ServiceFactory`. The `Application` would then access the factory through that global variable.

CONCLUSION

As we move forward in this book and cover higher-level architectural principles, the DIP will show up again and again. It will be the most visible organizing principle in our architecture diagrams. The curved line in Figure 11.1 will become the architectural boundaries in later chapters. The way the dependencies cross that curved line in one direction, and toward more abstract entities, will become a new rule that we will call the *Dependency Rule*.

1. In other words, the function that is invoked by the operating system when the application is first started up.