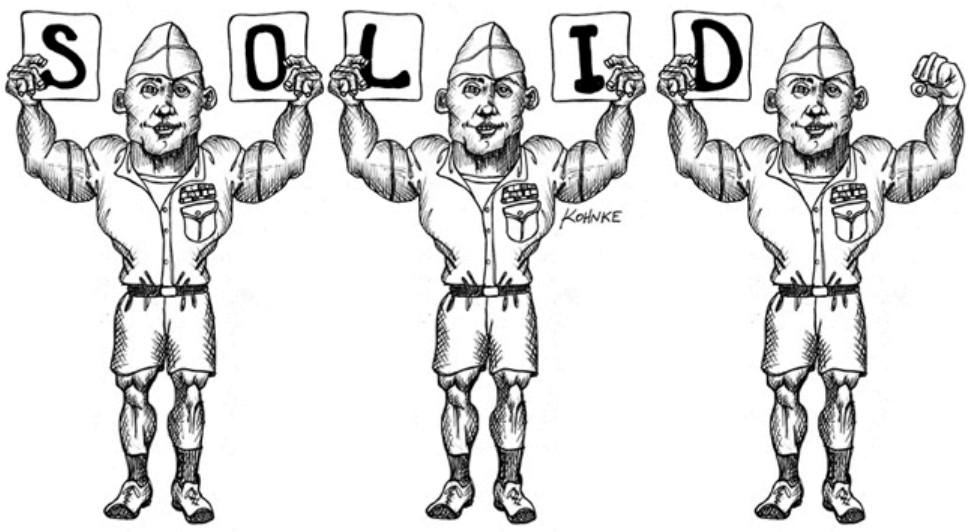
19

The SOLID Principles

This chapter is a significantly abridged and edited set of excerpts from my book Clean Architecture. *It is here to give you a quick reference to the principles mentioned elsewhere in the book, and to provide you with directions for further study.*



Good software systems begin with clean code. 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 only applicable 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
- Are the basis of components that can be used in other 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 functions and help 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, it is also possible to create a systemwide mess with well-designed mid-level components. So, once we have covered the SOLID principles, we will move on to their counterparts in the component world, and then, in Part III, to the principles of high-level architecture.

The history of the SOLID principles is long. I began to assemble them in the late ’80s while debating software design principles with others on Usenet (an early social media platform). 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, though I presented them in a different order.

Then, in 2004 or thereabouts, Michael Feathers sent me an email saying that if I rearranged the principles, their first letters of their initialisms would spell the word *solid*. And thus, the SOLID principles were born. The sections that follow describe each principle in detail. Here is the executive summary:

- **SRP**—The Single Responsibility Principle: The best structure for a software system is heavily influenced by the social structure of the organization that uses it, such that each software module has one, and only one, reason to change.
- **OCP**—The Open–Closed Principle: Bertrand Meyer made this famous in the ’80s. The gist is that in order 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 by changing existing code.

- **LSP**—The Liskov Substitution Principle: This is Barbara Liskov's famous definition of subtypes, from 1988. In short, this principle says that in order to build software systems from interchangeable parts, those parts must adhere to a contract that allows them 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: This principle says that the code that implements high-level policy should not depend on the code that implements low-level detail. Rather, low-level details should depend upon high-level policies.

These principles have been described in detail in many different publications[1] over the years. The sections that follow are not meant to replace those detailed writings. If you are not already familiar with these principles, what follows will be a good introduction, but insufficient for a deep understanding. You would be well advised to study them in the footnoted documents.

---

1. For example, [PPP02], http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod, and https://en.wikipedia.org/wiki/SOLID_(object-oriented_design) (or just google SOLID).

SRP: The Single Responsibility Principle

Of all the SOLID principles, this 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.

There *is* a principle like that, as we have seen in previous chapters. It is true 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 that principle 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 in order to satisfy users and stakeholders. So 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. This is because there will likely be more than one user or stakeholder who wants the system changed in the same way. So we're really referring to a group of one or more people who require changes. We shall refer to that group as an *actor*.

Thus, the final version of the SRP is:

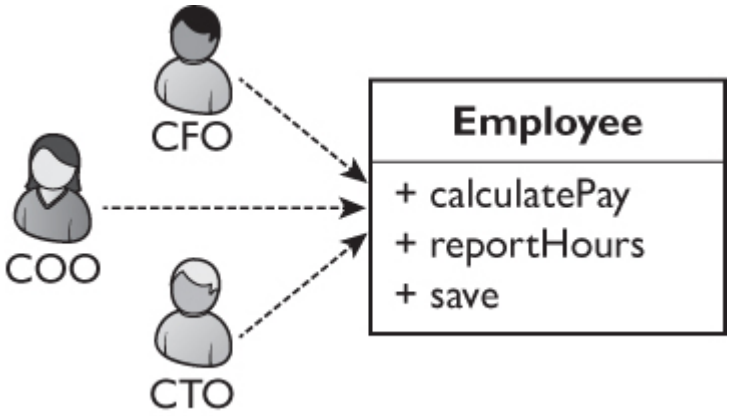
A module should be responsible to one, and only one, actor.

Now, what do we mean by the word *module*? 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 one of the many symptoms that result from violating it.

Accidental Duplication

Consider the `Employee` class from a payroll application. It has three methods: `calculatePay()`, `reportHours()`, and `save()`.

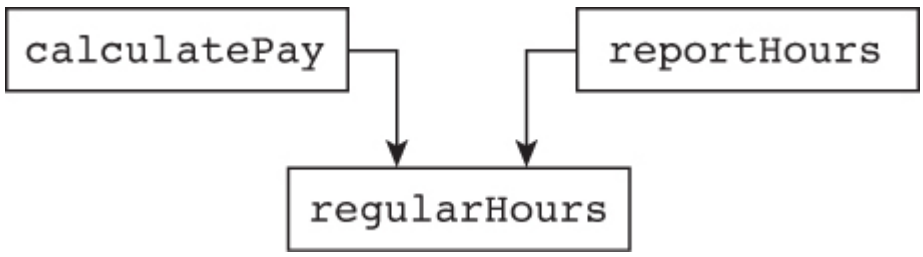


This class violates the SRP because those three methods are responsible to three very different actors.

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

For example, let's say that the `calculatePay()` function and the `reportHours()` function share a common algorithm for calculating nonovertime hours. Let's also say that the developers, who are careful not to duplicate code, put that algorithm into a function named `regularHours()`.



Now let's say that the CFO's team decides that the way nonovertime hours are calculated needs to be tweaked. But the COO's team in HR does not want that particular tweak, because they use nonovertime 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.

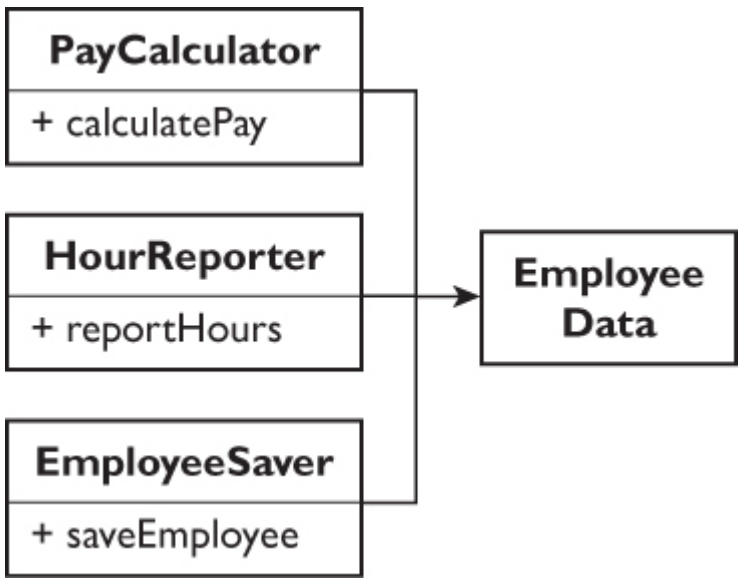
Of course, the COO's team doesn't know that this is happening. The COO's team continues to use the reports generated by the `reportHours()` function—but now they contain incorrect numbers. Eventually the problem is discovered, and the COO is livid because the bad data has cost his budget millions of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

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

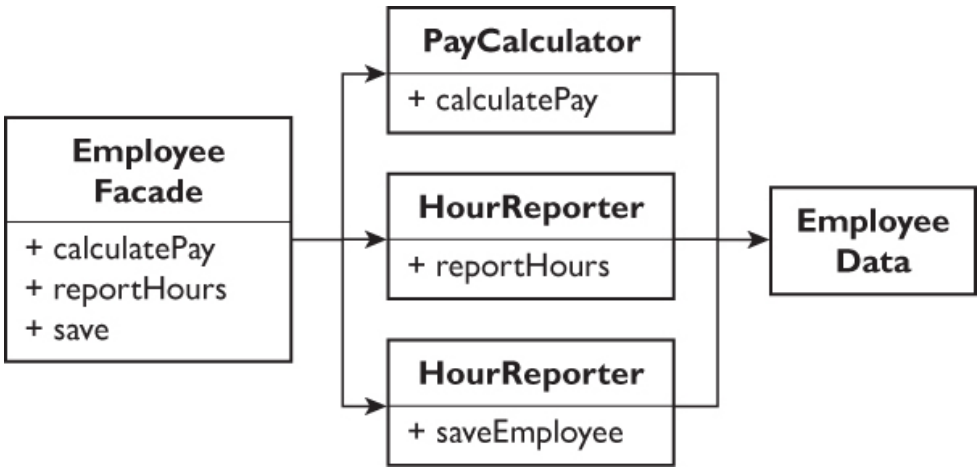
Solutions

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

Perhaps the most obvious way to do this is to separate the data from the functions. The three classes share access to `EmployeeData`, which is a simple data structure with no methods. 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.

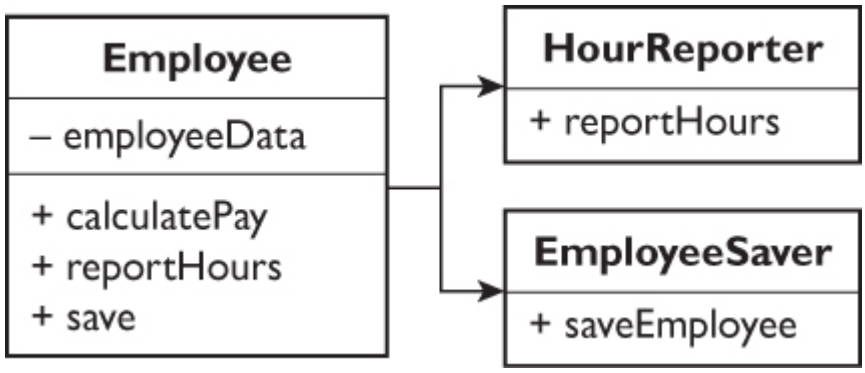


Of course, the downside of this solution is that the developers now have four classes that they have to instantiate and keep track of. A common solution to this dilemma is to use the Facade pattern.



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.



You might object to these solutions on the basis that every class would just contain 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 likely have many other public methods, and many more private methods in them.

Higher Levels

The SRP is about functions and classes. But the principle reappears in a different form at two more levels.

As we shall see, at the level of components it becomes the Common Closure Principle. And at the architectural level, it becomes the axis of

change responsible for the creation of architectural boundaries. We'll be studying all of this in the chapters to come.

OCP: The Open–Closed Principle

The OCP was coined in 1988 by Bertrand Meyer.[2] It says:

---

2. [OOSC], p. 23.

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 design. When simple extensions to the requirements force massive changes to the software, 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

A thought experiment will make this clear.

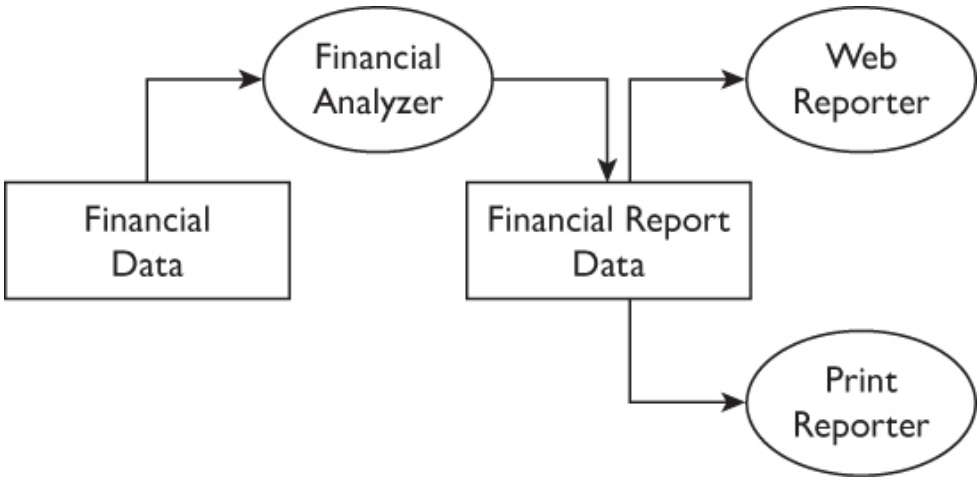
Imagine, for a moment, that we have a system that displays a financial summary on a web page. The data on the page is scrollable, and negative numbers are rendered in red.

Now imagine that the stakeholders ask that this same information be produced as 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 there will be some new code to write. But how much old code will have to change?

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 SRP), and then organizing the dependencies between those things properly (the DIP).

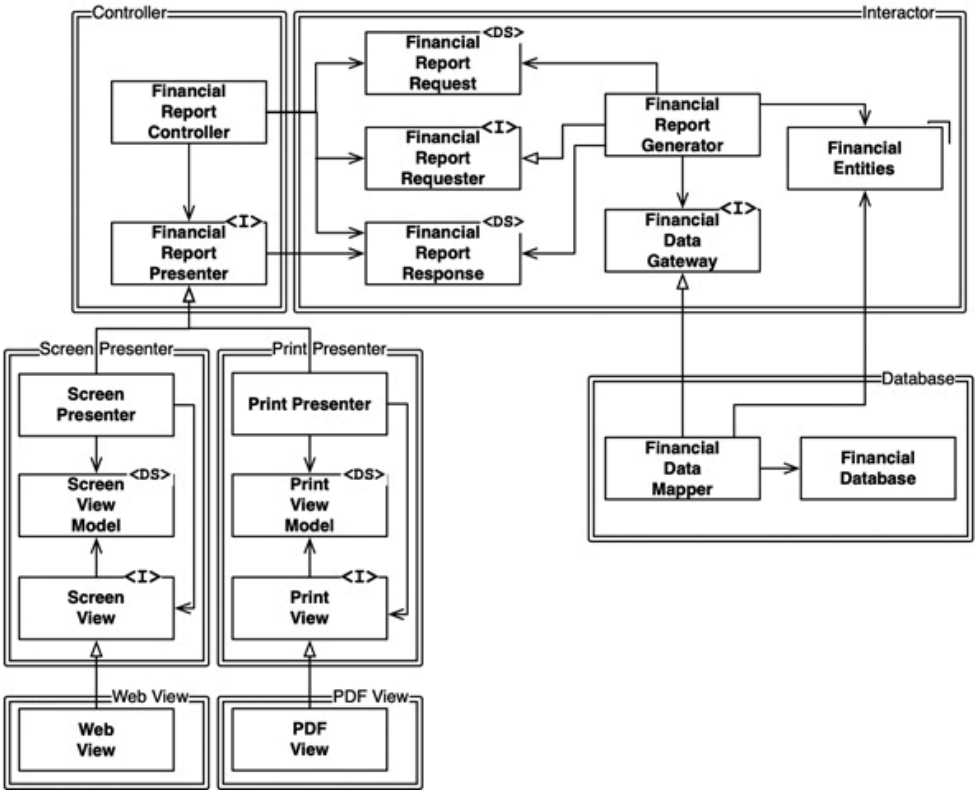


Applying the SRP, we might come up with the data-flow view above. Some analysis procedure inspects the financial data and produces reportable data, which is then formatted appropriately by the two reporter processes.

The essential insight here is that the report consists of three separate responsibilities: the calculation of the reported data and the presentation of that data into a web and printer form.

Having made this separation, we need to organize the source code dependencies in order to ensure that changes to one of those responsibilities do not cause changes in the other, and 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 following diagram.



Future Bob:

To be clear, I would never expect you to draw a diagram like this. I certainly don't when I'm thinking through issues like this. I might draw the abbreviated view that you see below on a whiteboard in order for the team to discuss it. Most of the classes within those components would be so obvious to the team that drawing them would be superfluous. If any team member had a particular question, someone else on the team could scribble a portion of the diagram on the whiteboard or on a napkin.

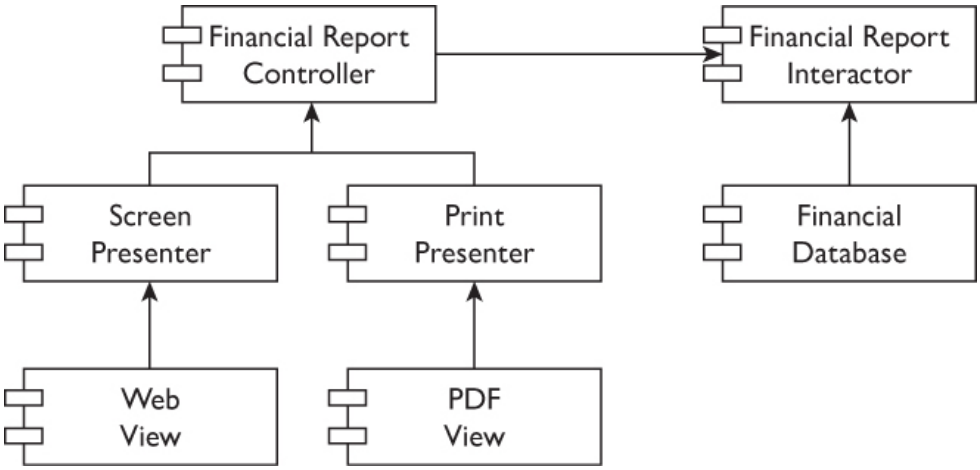
The component at the upper left is the Controller. On the upper right we have the Interactor. On the lower right there is the Database. And on the

lower left there are four components that represent the Presenters and the Views.

Classes marked with `<I>` are interfaces and 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. So, in the diagram above, `FinancialDataMapper` knows about `FinancialDataGateway` with an implements relationship, but `FinancialDataGateway` 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 following diagram.



These arrows point toward the components that we want to protect from change.

Let me say that again. If component A should be protected from changes in component B, then component B should depend upon 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—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 upon 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 upon 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 they 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 above, look again. Much of the complexity in that diagram was 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 in order 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 upon the `FinancialEntities`.

Transitive dependencies are a violation of the general principle that software entities should not depend upon things they don't directly use. We'll see that principle covered later when we talk about the ISP and the CRP.

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 design of systems. The goal is to make the system easy to extend without incurring a high impact of change. This is accomplished by partitioning the system into components and arranging them into a dependency hierarchy that protects higher-level components from changes in lower-level components.

LSP: The Liskov Substitution Principle

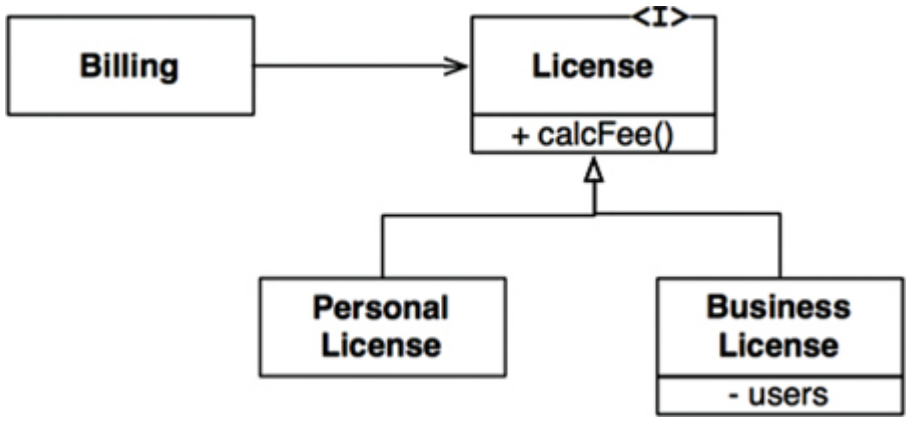
In 1988, Barbara Liskov wrote the following as a way of defining subtypes.

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.[3]

---

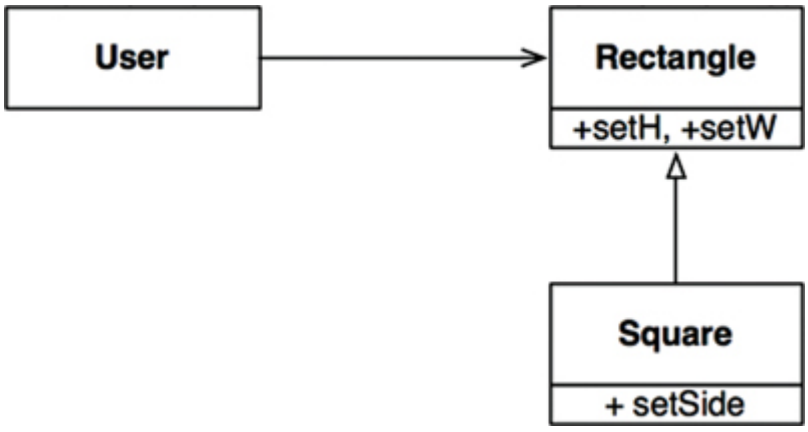
3. Barbara Liskov, "Data Abstraction and Hierarchy," *ACM* SIGPLAN Notices 23, no. 5 (May 1988): 17–34.

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



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

The canonical example of a violation of the LSP is the famed `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, but 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 … 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 detect 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.

Putting an `if` statement like that into the `User` violates the OCP. Thus, every violation of the LSP is a latent violation of the OCP.

LSP and Design

In the early years of the object-oriented revolution, we thought of this principle as a way to guide the use of inheritance, as shown in the previous section. However, over the years, we have morphed the LSP 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 upon well-defined interfaces and upon the substitutability of the implementations of those interfaces.

Taxi Aggregator

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

Let’s assume we 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 the taxi company. Once the customer decides, our system dispatches the taxi by using a RESTful service.

Now let's 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.

For example, let's say that 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/1530
  /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 let's say the Acme taxi company hired some programmers who didn't read the spec very carefully, and they abbreviated the destination field to just `dest`. And let's say that Acme was 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?

Clearly we'd 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 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 developer worth their 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 got very successful and bought the Purple Taxi company? What if they maintained the brand and the website but unified all their systems? Would we have to add another `if` statement for the word *purple*?

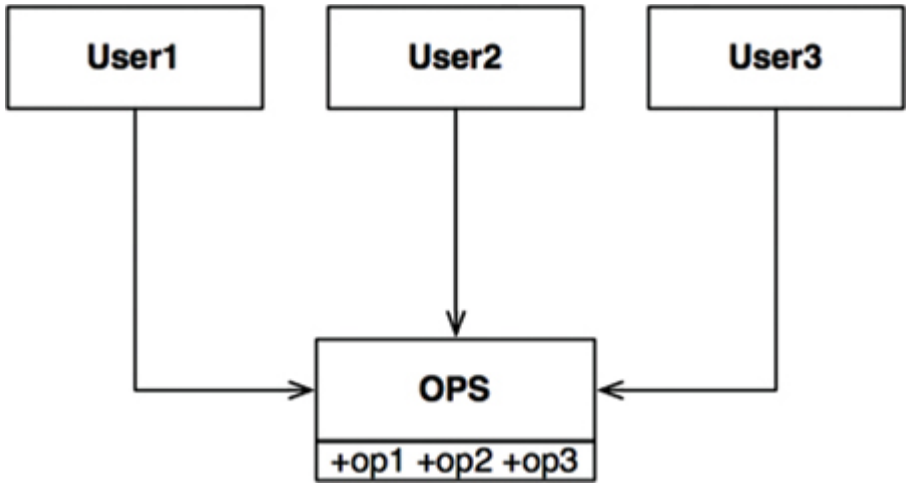
So, our architect would have to insulate the system from bugs like this by creating some kind of dispatch command creation module that was driven by a configuration database keyed by the dispatch URI. The configuration data might look something like this:

URI	Dispatch format
Acme.com	`/pickupAddress/%s/pickupTime/%s/dest/%s`
.	`/pickupAddress/%s/pickupTime/%s/destinatio`

The point of all this is that a simple violation of the LSP for a RESTful service can cause our architecture to be polluted with a significant number of extra mechanisms.

ISP: The Interface Segregation Principle

The ISP derives its name from the following:



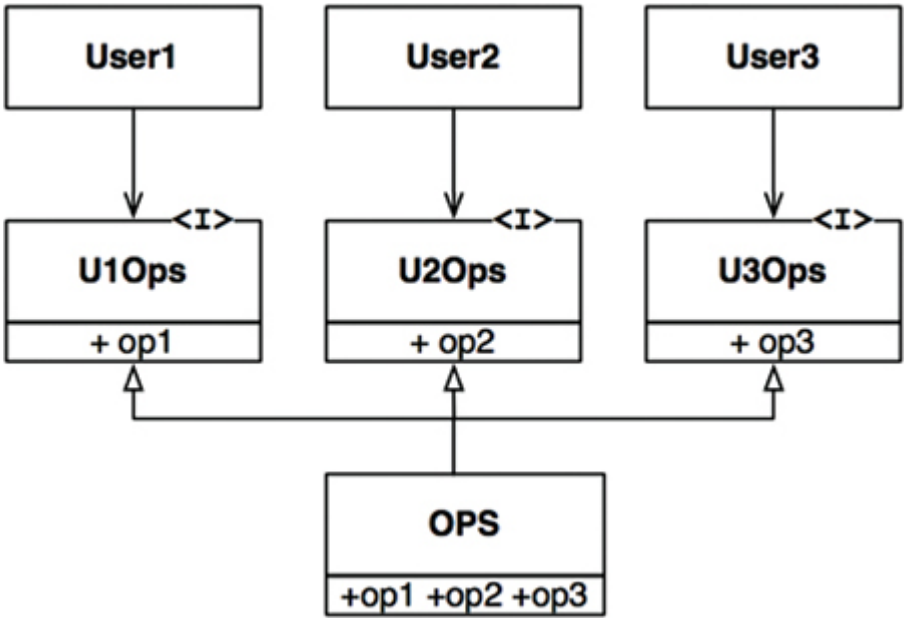
In the situation above, there are several users that use the operations of the `OPS` class. Let's assume that `User1` only uses `op1`, `User2` only uses `op2`, and `User3` only uses `op3`.

Now imagine that `OPS` is a class written in a language like C++.[4] Clearly, in that case, the source code of `User1` will inadvertently depend upon `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.

---

4. The situation in Java and C# is a bit more complicated. The compiler does not recompile modules based solely upon dates. Rather, it looks for changed declarations.

This can be resolved by segregating the operations into interfaces, as shown below.



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

ISP and Language

Statically typed languages like C++ force programmers to create declarations that users must `#include`. It is these included declarations in the source code that create the source code dependencies that can force recompilation and redeployment.

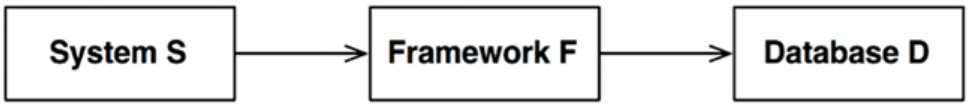
In languages like Java and C#, the dependencies are softer but not absent. The compilers follow the dependency tree using individual declarations rather than source files. So recompilation and redeployment are only necessary if declarations change.

In dynamically typed languages like Ruby and Python, interfaces don't exist in source code. Rather, they are inferred at runtime. Thus, there are no source code dependencies nor any interface declarations to force recompilation and redeployment. This is one reason that dynamically typed languages create systems that are more flexible and less coupled than statically typed languages.

This fact could lead you to conclude that the ISP is a language issue and not a design issue.

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

Consider, for example, a developer working on a system, `S`. He wants to include a certain framework, `F`, in the system. Now suppose that the authors of `F` have bound it to a particular database, `D`. So `S` depends upon `F`, which depends upon `D`.



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

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

The bottom line is:

Don't depend on things you don't need.

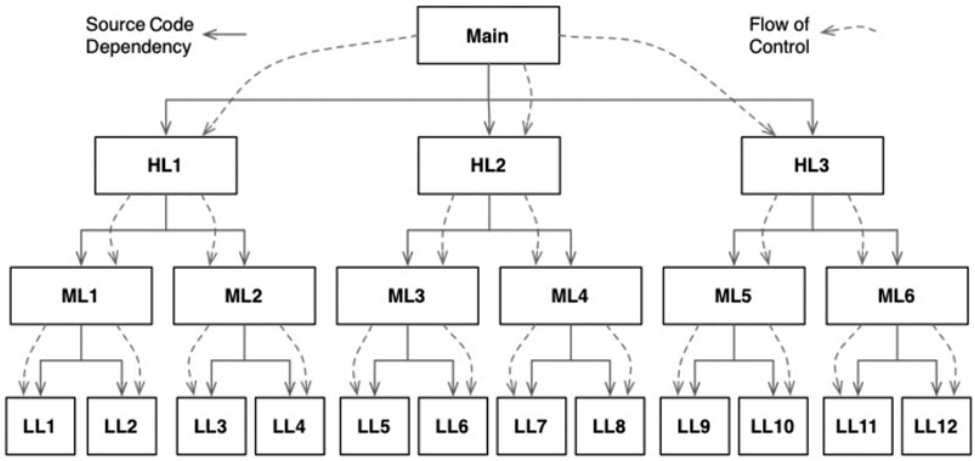
We'll see more about that when we discuss the Common Reuse Principle in Chapter 20, "Component Principles."

DIP: The Dependency Inversion Principle

"It's better to depend on something you control than on something you don't control, lest it end up controlling you."

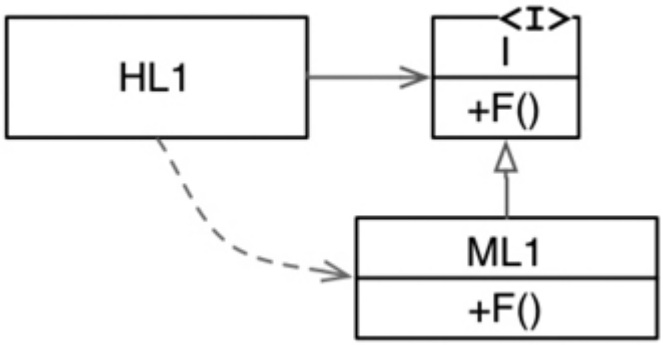
—James Grenning

In the ’60s and ’70s, virtually all programs were built according to the following pattern.



The source code dependencies followed the flow of control, without exception. High-level modules depended upon low-level modules because the high-level modules *called* the low-level modules.

OO gave us a different option: polymorphic interfaces.

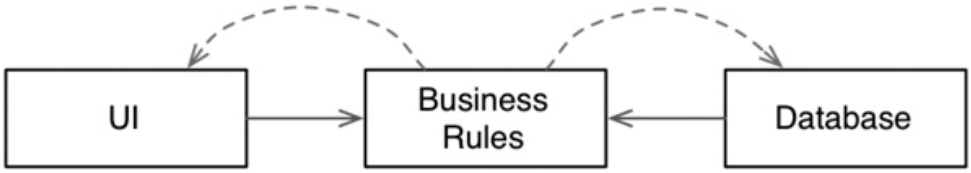


The insertion of an interface into the calling path inverts the source code dependency against the flow of control. In the above diagram, the high-level module `HL1` calls the `f` function in the lower-level module `ML1`; but it does so through the interface `I`. The source code dependency from `ML1` to `I` points *against* the flow of control.

The ability to invert dependencies in this way gives the software designer absolute control over every source code dependency in the system. If any are problematic (and they often are), they can be quickly inverted by inserting an interface.

This ability is power!

It allows us to create plug-in architectures that allow high-level policies, like the business rules, to be independent of low-level details, like the UI and the database.



In the above diagram, all source code dependencies (solid arrows) point from lower-level modules to higher-level modules. Both `UI` and `Database` depend upon and implement interfaces within `BusinessRules`. In other words, the concrete details depend upon abstractions. This 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`, or `include` statements should refer only to source modules containing interfaces, abstract classes, or some other kind of abstract declaration. Nothing concrete should be depended upon.

The same rule applies for dynamically typed languages, like Ruby or 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.

Treating this as a hard-and-fast rule is unrealistic. There are many concrete facilities that software systems must depend upon. For example, the `String` class in Java is concrete. It would be unrealistic to try to force it to be abstract. The source code dependency upon the concrete `java.lang.string` cannot, and should not, be avoided.

On the other hand, 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`.

And so we tend to ignore the stable background of operating system and platform facilities when it comes to the DIP. We tolerate those concrete dependencies because we know we can rely upon them not to change.

It is the *volatile* concrete elements of our system that we want to avoid depending upon. It is those 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. On the other hand, changes to concrete implementations do not always, or even usually, require changes to the interfaces that they implement. Therefore, interfaces are generally 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, therefore, is that stable software designs are those that avoid depending on volatile concretions, and 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, and 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.

While these are written as rules, they are actually more like warnings. A pragmatic programmer will violate this principle frequently. But a wise programmer will consider the warnings first.

Slavishly obeying this principle can lead to an explosion of interfaces that no one really needs. On the other hand, blithely ignoring this principle can cause your software to become tangled and rigid.

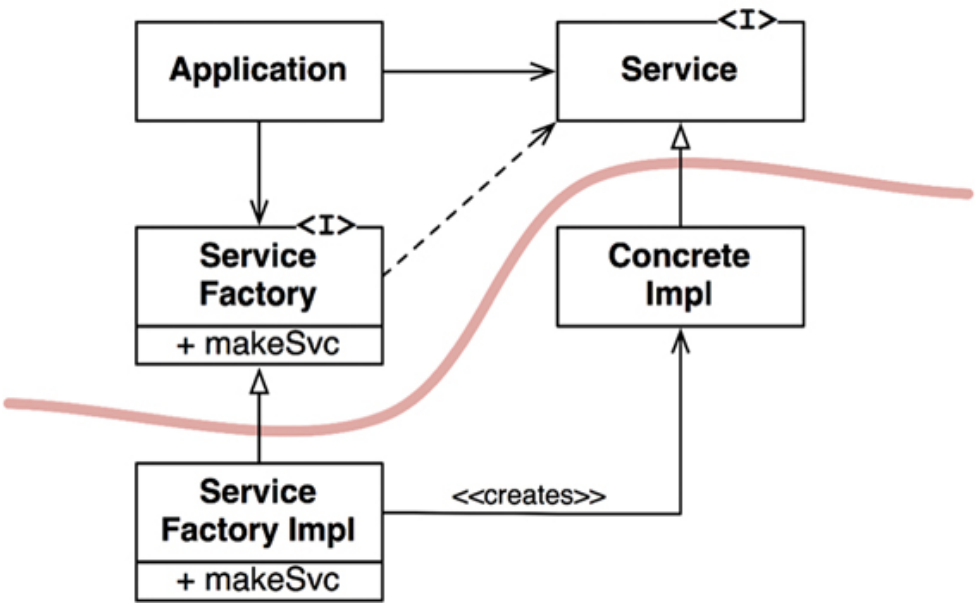
Finally, wise programmers will conform to this principle *gradually*, as the system evolves and the needs arise. With a test suite that they trust with their lives, they know that when interfaces need to be added in order to invert dependencies, the changes will be relatively easy to make and free of risk.

Factories

In order to comply with the DIP, the creation of volatile concrete objects may require special handling. This is because, in virtually all languages, the creation of an object requires a source code dependency upon the concrete definition of that object.

In most OO language, like Java, we would use an Abstract Factory[5] to eliminate this undesirable dependency.

5. [GOF95].



The preceding diagram 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 upon the `ConcreteImpl`, the `Application` calls the `makeSvc` method of the `ServiceFactory` interface. This method is implemented by the `ServiceFactoryImpl` class that derives from `ServiceFactory`. That implementation instantiates the `ConcreteImpl` and returns it as a `Service`.

Note the curved line. This is an architectural boundary. It separates the abstract from the concrete. All source code dependencies cross that line pointing in the same direction: toward the abstract side.

The 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 line in the opposite direction of the source code dependencies. The source code dependencies are inverted against the flow of control. That is why we refer to this principle as *dependency inversion*.

Concrete Components

The concrete component in the above diagram contains a single dependency that 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`[6] function. In the above case, 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.

---

6. That is, the function that is invoked by the operating system when the application is first started up.

As we proceed 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 the diagram above will become the architectural boundaries in later chapters. The way the dependencies cross that line in one direction and toward more abstract entities will become a new rule that we will call the Dependency Rule.